CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 1

1. (5 pts.) Problem 1

I understand the course policies.

2. (36 pts.) Problem 2

(a) $f = O(g)$: $\lim_{n\to\infty} \frac{6n\cdot 2^n + n^{100}}{3^n} = \lim_{n\to\infty} \frac{6n}{1.5^n} + \lim_{n\to\infty} \frac{n^{100}}{3^n} = 0$, now since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, we have $f = O(g)$.

(b) $f = \Theta(g)$: $\lim_{n\to\infty} \frac{\log 2n}{\log 3n} = \lim_{n\to\infty} \frac{\log 2 + \log n}{\log 3 + \log n} = 1$.

(c) $f = \Omega(g)$: $\lim_{n\to\infty} \frac{\sqrt{n}}{\sqrt[3]{n}} = \infty$

(d) $f = \Omega(g)$: $\lim_{n\to\infty} \frac{\frac{n^2}{\log n}}{n(\log n)^4} = \lim_{n\to\infty} \frac{n}{(\log n)^5} = \infty$

(e) $f = \Theta(g)$: n^2 dominates both $n\log n$, and $(\log n)^5$ terms, which implies that $f = \Theta(n^2), g = \Theta(n^2)$,

(f) $f = O(g)$: Let $k = \log_2 n$. Then $f(n) = k^k$ and $g(n) = 2^{k^2} = (2^k)^k$. Note that $2^k \geq k$ for all $k \geq 1$. Therefore, $(2^k)^k \geq k^k$ for all $k \geq 1$ and $f = O(g)$. Alternatively, taking the log of both functions (a valid manipulation since logarithms are increasing functions) gives $\log_2 f(n) = (\log_2 n)(\log_2 \log_2 n)$ and $\log_2 g(n) = (\log_2 n)^2$. Because $\log_2 \log_2 n \leq \log_2 n$, $(\log_2 n)(\log_2 \log_2 n) \leq (\log_2 n)^2$.

(g) $f = \Theta(g)$: We can write $f = n\log(n^{20}) = 20n\log n = \Theta(n\log n)$, and $g = \log(3n!) = \Theta(\log(n!))$. Thus, it remains to show that $\log(n!) = \Theta(n\log n)$, but we know this is true, as we saw the proof in Worksheet 1, problem 2(i).

(h) $f = \Theta(g)$: we can write $8\log n = \log(n^8) < \log(n^9 + \log n) < \log(n^{10}) = 10\log n \Rightarrow f = \Theta(\log n)$ and $g = \log 2n = \log 2 + \log n = \Theta(\log n)$. Another possible set of bounds for f are $\log(n^9) < \log(n^9 + \log n) < \log(2n^9)$.

(i) $f = \Omega(g)$: Let $k = \lfloor \sqrt{n} \rfloor$, then we have $f(k) \geq 8^{k^2}\cdot k^4$, $g(k) = k!$. Now, note that $8^k \geq k$ for all $k \geq 1$, and so $8^{k^2} = (8^k)^k \geq k^k$ for all $k \geq 1$. From Worksheet 1, we know that $k^k \geq k!$ and so $8^{k^2} \geq k!$ which also implies that $8^{k^2}\cdot k^4 \geq k!$ for all $k \geq 1$. Thus, it follows from the definition of O that $k! = O(8^{k^2}\cdot k^4)$ (taking $c = 1$ and $n_0 = 1$); it follows from the definition of Omega that $f = \Omega(g)$.

3. (15 pts.) Problem 3

By the formula for the sum of a partial geometric series, for $c \neq 1$: $S(k) := \sum_{i=0}^{k} c^i = \frac{1-c^{k+1}}{1-c}$. Thus,

- If $c > 1$, we have $c^{k+1} > 1$, and then $S(k) = \frac{c^{k+1}-1}{c-1}$. Now we can write

$$\lim_{k\to\infty} \frac{S(k)}{c^k} = \lim_{k\to\infty} \frac{c - \frac{1}{c^k}}{c-1} = \frac{c}{c-1} - \frac{1}{c-1}\lim_{k\to\infty} \frac{1}{c^k}.$$

  Since $c > 1$, we can conclude $\lim_{k\to\infty} \frac{1}{c^k} = 0$. Therefore we have $\lim_{k\to\infty} \frac{S(k)}{c^k} = \frac{c}{c-1}$. Note that $0 < \frac{c}{c-1} < \infty$, therefore $S(k) = \Theta(c^k)$.
- If $c = 1$, then every term in the sum is 1. Thus, $S(k) = k + 1 = \Theta(k)$.

- If $c < 1$, then $\frac{1}{1-c} > \frac{1-c^{k+1}}{1-c} = S(k) > 1$. Thus, $S(k) = \Theta(1)$.

4. (20 pts.) Problem 4

(a) $f(n)$ is a degree d polynomial. We have $\lim_{n\to\infty} \frac{f(n)}{n^k} = 0$, for $k > d$. So, we can conclude that for $k \geq d$, $f(n) = O(n^k)$. On the other hand, for $k < d$, $\lim_{n\to\infty} \frac{f(n)}{n^k} = \infty$, which means that $f(n) = \Omega(n^k)$ in this case. Also for $k = d$, we have $\lim_{n\to\infty} \frac{f(n)}{n^k} = a_d$, so $f(n) = \Theta(n^k)$

(b) Part (b) is a special case of part (c); same proof works. Alternatively, this problem can also be solved by remembering that $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$.

(c) Since $k \leq n$ every term in the sum is at most n, so

$$\sum_{k=1}^{n} k^j = 1^j + \cdots + n^j \leq n^j + \cdots + n^j = \sum_{k=1}^{n} n^j = n^{j+1}.$$

We do something similar for the lower bound. The only additional idea is that we only look at the second half of the sum. The smallest element in the second half of the sum corresponds to $k = n/2$ (assuming without loss of generality that n is even). Then,

$$\sum_{k=1}^{n} k^j \geq \sum_{k=n/2}^{n} k^j \geq \sum_{k=n/2}^{n} (\tfrac{n}{2})^j = (\tfrac{n}{2})^{j+1} = \frac{1}{2^{j+1}} \cdot n^{j+1}.$$

(d) First we show the upper bound. $\sum_{i=1}^{n} \sum_{j\neq i, j=1}^{n} ij \leq (\sum_{i=1}^{n} i)^2 = (\frac{n(n+1)}{2})^2 = O(n^4)$. For the lower bound, we can write:

$$\sum_{i=1}^{n} \sum_{j\neq i, j=1}^{n} ij \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} i \sum_{j=\lceil \frac{n}{2} \rceil + 1}^{n} j \geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \frac{n}{2} \sum_{j=\lceil \frac{n}{2} \rceil + 1}^{n} (\frac{n}{2} + 1) \quad \geq \tfrac{n}{2}(\tfrac{n}{2} + 1) \cdot (\frac{n}{2})^2 \geq \frac{n(n+2)}{4} \cdot \frac{n^2}{4} = \Omega(n^4).$$

Alternatively, the summation can be expanded into $(\sum_{i=1}^{n} i)^2 - \sum_{i=1}^{n} i^2$, then simplified using the standard summations from 1 to n. This will give a polynomial in n with degree 4, which has been shown to be $\Theta(n^4)$. The intuition for this approach is that the only thing stopping the original summation from being easily decoupled is the condition $j \neq i$, which excludes the same products as the new i^2 summation subtracts.

5. (24 pts.) Prove the statements

1. Based on the definition, $f(n) = O(g(n))$ means that there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. Thus, $0 \leq \frac{1}{c} f(n) \leq g(n)$ for all $n \geq n_0$. As $\frac{1}{c}$ is positive, given the definition of the Ω-notation, we can deduce that $f(n) = O(g(n))$ means the same as $g(n) = \Omega(f(n))$.
2. According to the definition of Θ-notation, we need to find two constants c_1 and c_2 such that $0 \leq c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n))$ for all $n \geq n_0$ where n_0 is a positive constant. Given that $f(n)$ and $g(n)$ are asymptotically non-negative, we can make the following conclusions:
   - $\max(f(n), g(n)) \leq f(n) + g(n)$
   - $\max(f(n), g(n))$ comprises at least half of $f(n) + g(n)$, meaning $\max(f(n), g(n)) \geq 0.5(f(n) + g(n))$
   - $0.5(f(n) + g(n)) \geq 0$

   As such, c_1 could be 0.5 and c_2 could be 1. Therefore, $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

3. We know that one can directly translate between logarithms of different bases using the following fundamental identity:

$$\log_a n = \frac{\log_b n}{\log_b a}$$

So we can get $\log_a n = \frac{1}{\log_b a} \log_b n$, where $\log_b a$ is a constant. Therefore, $\log_a n = \Theta(\log_b n)$

Rubric:

Problem 1

Assign full credit if "I understand the course policies" is written. Subtract one point if they forgot to mention their collaborators or write "none" for the collaborators.

Problem 2

Each part is 4 pts: 2 pt for identifying right relation between f and g and 2 pts for a reasonable explanation.

Problem 3
Case $c > 1$: 6 pts
Case $c = 1$: 3 pts
Case $c < 1$: 6 pts

Problem 4

Each part is worth 5 pts.
part a: showing only one case is worth 2.5 pts.
part b-c-d: showing only upper bound is worth 2 pts, and showing only lower bound is worth 3 pts.

Problem 5

1. This part is worth 8 points.
   5 points: the inference from $0 \leq f(n) \leq cg(n)$ to $0 \leq \frac{1}{c} f(n) \leq g(n)$
   3 points: emphasize $\frac{1}{c}$ is positive

2. This part is worth 8 points.
   3 points: correctly prove the upper bound
   4 points: correctly prove the lower bound
   1 point: emphasize the lower bound i.e. $c_1(f(n) + g(n))$ is non-negative

3. This part worth 8 points, students should get 4 point if they show $\log_a n = \frac{\log_b n}{\log_b a}$, and get the rest 4 point if they show $\log_a n = \frac{1}{\log_b a} \log_b n \leq c \log_b n$

CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 2

1. (20 pts.) Problem 1

(a) This is false. A counterexample is $f(n) = 2n$ and $g(n) = n$. $f(n)$ is $O(g(n))$ in this case because we can find constants $c = 3$ and $n_0 = 1$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. However, since $2^{f(n)} = 2^{2n}$ and $2^{g(n)} = 2^n$, $\lim_{n\to\infty} \frac{2^{2n}}{2^n} = \infty \neq 0$. So, $2^{f(n)}$ grows faster than $2^{g(n)}$ asymptotically. Thus, the statement is false.

(b) This is true. Proof: As $f(n)$ is $O(g(n))$, there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. Then, $0^2 \leq f(n)^2 \leq c^2 g(n)^2$ for all $n \geq n_0$. Let $d = c^2$. We have $0 \leq f(n)^2 \leq dg(n)^2$ for all $n \geq n_0$. So, we've found constants d and n_0 that satisfy the definition of Big-O notation. Thus, $f(n)^2$ is $O(g(n)^2)$.

(c) This is false. A counterexample is $f(n) = 2n$ and $g(n) = n$. $f(n)$ is $O(g(n))$ in this case because we can find constants $c = 3$ and $n_0 = 1$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. Let $h(n)$ be $0.5n$. $h(n)$ is $O(g(n))$ in this case because we can find constants $c = 3$ and $n_0 = 1$ such that $0 \leq h(n) \leq cg(n)$ for all $n \geq n_0$. However, since $2^{f(n)} = 2^{2n}$ and $2^{h(n)} = 2^{0.5n}$, $\lim_{n\to\infty} \frac{2^{2n}}{2^{0.5n}} = \infty \neq 0$. So, $2^{f(n)}$ grows asymptotically faster than $2^{h(n)}$, which is an example of $2^{O(g(n))}$. Thus, the statement is false.

(d) This is false. A counterexample is $f(n) = 2(1+\frac{1}{n})$ and $g(n) = 1+\frac{1}{n}$. $f(n)$ is $O(g(n))$ in this case because we can find constants $c = 2$ and $n_0 = 1$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. However, $\log_2 f(n) = \log_2(2(1+\frac{1}{n})) = \log_2 2 + \log_2(1+\frac{1}{n}) = 1 + \log_2(1+\frac{1}{n})$ and $\log g(n) = \log_2(1+\frac{1}{n})$. Note that $\lim_{n\to\infty}(1+\frac{1}{n}) = 1$ implies that $\lim_{n\to\infty} \log(1+\frac{1}{n}) = 0$. Any constant multiple $c \log g(n)$ also goes to 0 as n grows, while $\log f(n)$ always goes to 1 as n grows. Therefore, $\log f(n) \neq O(\log g(n))$.

2. (20 pts.) Problem 2

(a) We can observe that term i with coefficient a_i has $\log_2 i$ multiplications. Overall, there are $\sum_{i=1}^{n} \log_2 i = \log_2(n!) = O(n \log n)$ multiplications (from worksheet 1 problem 2-i, $\log(n!) = \Theta(n \log n)$). There are n additions, thus $O(n)$.

(b) If we consider the value of z *after* each iteration we obtain

$$\begin{aligned}
i = n-1 \quad &\rightarrow \quad z = a_n x_0 + a_{n-1} \\
i = n-2 \quad &\rightarrow \quad z = a_n x_0^2 + a_{n-1} x_0 + a_{n-2} \\
i = n-3 \quad &\rightarrow \quad z = a_n x_0^3 + a_{n-1} x_0^2 + a_{n-2} x_0 + a_{n-3} \\
&\cdots \\
i = 0 \quad &\rightarrow \quad z = a_n x_0^n + a_{n-1} x_0^{n-1} + \cdots + a_1 x_0 + a_0,
\end{aligned}$$

To describe in more detail, since the coefficients a_i are added to z in order from n to 0, the term with coefficient a_i multiplies with x_0 a total of i times. So, we have the desired polynomial $a_0 + a_1 x_0 + a_2 x_0^2 + \cdots + a_n x_0^n$ at the end.

(c) Every iteration of the for loop uses one multiplication and one addition, so the routine uses n additions and n multiplications.

3. (20 pts.) Problem 3

(a) The while loop in algorithm 2 takes $y-1$ iterations, and in each iteration we have to compute $z \cdot x$. Note that the multiplication of an n_1 bit number by an n_2 bit number results in a number with at most n_1+n_2 bits. Therefore, at i^{th} iteration of the while loop, z has $O(in)$ number of bits (before multiplication), therefore the cost of multiplication at i^{th} iteration is $O(ni\log(ni))$, and since there are $y-1$ iterations, the total running time would be:

$$\sum_{i=1}^{y-1} O(ni\log(ni)) = O(n)\sum_{i=1}^{y-1} i\log(ni) \tag{1}$$

$$= O(n)(\sum_{i=1}^{y-1} i\log n + \sum_{i=1}^{y-1} i\log i) \tag{2}$$

$$= O(n\log n)\sum_{i=1}^{y-1} i + O(n)\sum_{i=1}^{y-1} i\log i \tag{3}$$

$$= O(y^2 n\log n) + O(ny^2\log y) \tag{4}$$

$$= O(n^2y^2) \tag{5}$$

(b) The function recurses until y becomes 0, dividing y by 2 each time. Therefore, there are $O(\log_2 y) = O(m)$ recursive calls. In each iteration, we either compute $z \cdot z$ or $z \cdot z \cdot x$. Both cases have a constant number of multiplications. Here, because i is $O(m)$ instead of $O(y)$, z has $O(n2^i)$ bits. Each multiplication then takes $O(n2^i\log(n2^i))$ time. Therefore, the total running time is:

$$\sum_{i=1}^{m} O(n2^i\log(n2^i)) = O(n)\sum_{i=1}^{m} 2^i\log(n2^i) \tag{6}$$

$$= O(n)(\sum_{i=1}^{m} 2^i\log n + \sum_{i=1}^{m} 2^i\log 2^i) \tag{7}$$

$$= O(n\log n)\sum_{i=1}^{m} 2^i + O(n)\sum_{i=1}^{m} i2^i \tag{8}$$

$$\leq O(n\log n)\sum_{i=1}^{m} 2^i + O(mn)\sum_{i=1}^{m} 2^i \tag{9}$$

$$= O(2^m n\log n) + O(2^m mn) \tag{10}$$

$$= O(yn\log n) + O(yn^2) \tag{11}$$

$$= O(n^2y) \tag{12}$$

Where step (10) uses the growth rate of the sum of a geometric series proven in homework 1 problem 3 and step (11) uses the stated assumption that $n \geq m$.

4. (20 pts.) Problem 4

(a) For any 2×2 matrices X and Y:

$$XY = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix} = \begin{pmatrix} x_{11}y_{11}+x_{12}y_{21} & x_{11}y_{12}+x_{12}y_{22} \\ x_{21}y_{11}+x_{22}y_{21} & x_{21}y_{12}+x_{22}y_{22} \end{pmatrix}$$

This shows that every entry of XY is the addition of two products of the entries of the original matrices. Hence every entry can be computed in 2 multiplications and one addition. The whole matrix can be calculated in 8 multiplications and 4 additions.

(b) Let $A' = XA$, where A is an arbitrary 2×2 matrix, and $X = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Since entries of A' are only the sum of at most two entries of A, then we can say that number of bits of A' entries are at most one bit more than number of bits in A. Therefore, each time we multiply a matrix by X, the number of bits of each entry only increases by at most one bit. Since all the entries of X can be stored in 1 bit, we can conclude that the entries of X^i have as most i bits. $i < n$ implies that the number of bits is $O(n)$.

Alternatively, we can use the fact that X^i stores the Fibonacci numbers: $X^i = \begin{pmatrix} F_{i-1} & F_i \\ F_i & F_{i+1} \end{pmatrix}$, which are known to grow more slowly than 2^i (see worksheet 2). The largest element, F_{i+1} has at most $O(log_2(2^{i+1})) = O(i) = O(n)$ bits.

(c) In each call of `matrix`(X,n), we will go from n to $\frac{n}{2}$. So, it takes $\lfloor \log_2 n \rfloor$ recursive calls for the algorithm to end, and to return the output. Also, in each call, we have to do either $Z \cdot Z$ or $Z \cdot Z \cdot X$. Now, note that in i^{th} iteration or i^{th} recursive call we have $Z = X^{\frac{n}{2^i}}$, which means that entries of Z at i^{th} recursive call has at most $\frac{n}{2^i} = O(n)$ bits (According to part b). Now note that for computing either $Z \cdot Z$ or $Z \cdot Z \cdot X$, we have a constant number of multiplications and additions between numbers with $O(n)$ bits, which takes $O(M(n))$ time. Therefore, the total run time is:

$$\sum_{i=1}^{\lfloor \log_2 n \rfloor} O(M(n)) = O(M(n) \log n) \tag{13}$$

5. (20 pts.) Problem 5

(a) The answer is at least 2^h and at most $2^{h+1} - 1$. This is because a complete binary tree of height $h-1$ has $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ elements, and the number of elements in a heap of depth h is strictly larger than the number of vertices in a complete binary tree of height $h-1$ and less than (or equal) the number of nodes in a complete binary tree of height h.

(b) The array is not a Min heap. The node containing 26 is at position 9 of the array, so its parent is at position 4, which contains 35. This violates the Min Heap Property.

(c) Consider the min heap with n vertices where the root and every other node contains the number 2. Suppose now that 1 is inserted to the first available position at the lowest level of the heap. That is, $A[i] = 2$ for $0 \leq i \leq n-1$ and $A[n] = 1$. Since 1 is the minimum element of the heap, when Heapify-UP is called from position n, the node containing 1 must be swapped through each level of the heap until it is the new root node. Since the heap has height $\lfloor \log n \rfloor$, Heapify-UP has worst-case time $\Omega(\log n)$.

Rubric:

Problem 1, total points 20

(a) 5 points.
2 point: correct conclusion (statement is false)
1.5 points: a counterexample that makes sense
1.5 points: an explanation of why the provided counterexample shows the statement is false

(b) 5 points.
2 point: correct conclusion (statement is true)
3 points: a proof that reasons by making an association with the definition of big-O notation

(c) 5 points.
2 point: correct conclusion (statement is false)
1.5 points: a counterexample that makes sense
1.5 points: an explanation of why the provided counterexample shows the statement is false

(d) 5 points.
2 point: correct conclusion (statement is false)
1.5 points: a counterexample that makes sense
1.5 points: an explanation of why the provided counterexample shows the statement is false

Problem 2, total points 20

(a) 5 points.
1.5 points: correct answer for the number of sums
1.5 points: correct answer for the number of multiplications
2 points: the explanations make sense
Note: answers with exact numbers only or in big-O notations only are both acceptable, as long as the provided exact numbers or the big-O notations are correct.

(b) 10 points.
3 points: provide a proof
7 points: describe the pattern of the loop in the proof

(c) 5 points.
1.5 points: correct answer for the number of sums
1.5 points: correct answer for the number of multiplications
2 points: the explanations make sense
Note: answers with exact numbers only or in big-O notations only are both acceptable, as long as the provided exact numbers or the big-O notations are correct.

Problem 3, 20 pts

(a) 10 points. 6 points for showing the runtime of each iteration is $O(ni\log(ni))$. 4 points for manipulating the summation to show the overall runtime is $O(n^2y^2)$.

(b) 10 points.
2 points: shows $O(m) = O(\log_2 y)$ recursive calls.
4 points: shows the runtime of each recursive call is $O(n2^i\log(n2^i))$
4 points: manipulating the summation to show the overall runtime is $O(n^2y)$ (2 points for $O(n^3y)$).

Problem 4, 20 pts

(a) part a is worth 3 pts. Computing the multiplication of two matrices correctly is worth 2 pts, even if the final answer is not correct.

(b) part b is worth 5 pts.

(c) part c is worth 12 pts. 3 pts, for showing that there are $O(\log_2 n)$ recursive call, and 9 pts for showing the runtime for each recursive call which is $O(M(\frac{n}{2^i}))$. However, $O(M(n))$ is also accepted for the runtime of each recursive call.

Problem 5, 20 pts.

(a) 5 points for this part, if at least 2^h is pointed, 1.5 pts, if at most $2^{h+1} - 1$ is pointed, 1.5 pts, reasonable explanation 2 pts

(b) answer as No, 2.5 pts, denote that element 26 is in the wrong place, and it should be swapped with 35, 2.5 pts

(c) any clear and reasonable explanation should be 10 pts

CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 3

1. (20 pts.) Problem 1

(a) After Build-Heap: $[1,5,2,7,6,4,3,9,8,10]$
After 1st iteration of the loop: $[2,5,3,7,6,4,10,9,8,1]$
After 2nd iteration of the loop: $[3,5,4,7,6,8,10,9,2,1]$
After 3rd iteration of the loop: $[4,5,8,7,6,9,10,3,2,1]$

(b) Proof: Assume the element at position $\lfloor \frac{n}{2} \rfloor$ is an internal node, not a leaf. Then, its left child should be at position $2 \cdot \lfloor \frac{n}{2} \rfloor + 1 \geq 2 \cdot \frac{n}{2} - 1 + 1 = n > n-1$, which is the index of the last element in heap. Therefore, there is a contradiction. Thus, the element at position $\lfloor \frac{n}{2} \rfloor$ can't have any children, meaning it must be a leaf. The same reasoning can be applied to other positions in the statement since they are even larger than $\lfloor \frac{n}{2} \rfloor$. So, the nodes at positions $\lfloor \frac{n}{2} \rfloor, \ldots, n-1$ must be the leaves.

(c) Consider an array $[2,3,4,1]$. If we increase i from 0 to $\lfloor \frac{n}{2} \rfloor - 1$, there is no swap in the first iteration. In the second iteration, 3 swaps with 1. Then, the loop and the Build-Heap procedure are finished with the array $[2,1,4,3]$ which doesn't satisfy the min heap property since 2 is greater than its left child 1. So, increasing i from 0 to $\lfloor \frac{n}{2} \rfloor - 1$ can't guarantee that the array we end up obtaining from Build-Heap has the min heap property.

2. (15 pts.) Problem 2

We keep $M+1$ counters, one for each of the possible values of the array elements. We can use these counters to compute the number of elements of each value by a single $O(n)$-time pass through the array. Then, we can obtain a sorted version of x by filling a new array with the prescribed numbers of elements of each value, looping through the values in ascending order. This algorithm is called Counting Sort. Notice that the $\Omega(n \log n)$ bound does not apply in this case, as this algorithm is not comparison-based.

3. (15 pts.) Problem 3

(a) We get the recurrence $T(n) = T(n/2) + \Theta(1)$ for the running time of Binary Search. Applying the master theorem with $a = 1$, $b = 2$, $d = 0$ and $\log_b a = 0$, we note that we are in the second case, and so $\Theta(n^{\log_2 1} \log(n)) = \Theta(\log(n))$

(b) For Ternary Search the recurrence is $T(n) = T(n/3) + \Theta(1)$. Applying the master theorem with $a = 1$, $b = 3$, and $d = 0$, we have that $\log_3 1 = 0 = d$. So, we are in the second case, and we have that $\Theta(n^{\log_3 1} \log(n)) = \Theta(\log(n))$

4. (20 pts.) Problem 4

Let $m = \lfloor \frac{n}{2} \rfloor$, first we divide the input sequence in two halves ($A_1 = a_1, \ldots, a_m$ and $A_2 = a_{m+1}, \ldots, a_n$). Also, let $T(n)$ be the time it takes to sort the input sequence of n numbers and to count the number of inversions. First, we recursively sort, and count each half, which takes $2T(\frac{n}{2})$. Then, we should merge two halves, and count the number of inversions between the two halves. We can do this as follows. Remember that the two halves are A_1 and A_2, and let B_1, and B_2, be the output of recursive call on A_1 and A_2. Let i be the pointer to array B_1, and j be the pointer to array B_2. Also initialize i and j to zero (for example $i = 0$ points to first element of array B_1). (Merge part:) Compare the first elements of B_1, and B_2, and increase the pointer of the smaller element by one, and put it in output array S.(Count part:) Also, whenever we increase the B_2

pointer by one, we increase the total number of inversions by $\texttt{length}(B_1) - i$, since whenever we increase j (or B_2 pointer) by one, $B_2[j]$ should be less than $B_1[i : \frac{n}{2}]$, which makes $\texttt{length}(B_1) - i$ inversions. To get more details, You can see the pseudocode in algorithm 1.

Algorithm 1 sort, and count

Input: an array $A = [a_1, a_2, \ldots, a_n]$, and its length n
function SORT-COUNT(A, n)
  if $n == 1$ **then** return $A[0], 0$
  end if
  $A_1 \leftarrow [a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}]$, $A_2 \leftarrow [a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n]$
  $B_1, cnt_1 \leftarrow \texttt{sort-count}(A_1, \lfloor \frac{n}{2} \rfloor)$
  $B_2, cnt_2 \leftarrow \texttt{sort-count}(A_2, \lceil \frac{n}{2} \rceil)$
  $i \leftarrow 0$
  $j \leftarrow 0$
  $S \leftarrow$ an array of length n
  $k \leftarrow 0$
  $cnt \leftarrow 0$
  while $i < \texttt{length}(B_1)$ and $j < \texttt{length}(B_2)$ **do**
    if $B_2[j] \geq B_1[i]$ **then**
      $S[k] = B_1[i]$, $i \leftarrow i + 1$
    else
      $S[k] = B_2[j]$, $j \leftarrow j + 1$, $cnt \leftarrow cnt + \texttt{length}(B_1) - i$
    end if
    $k \leftarrow k + 1$
  end while
  if $i = \texttt{length}(B_1)$ **then**
    $S[k : n-1] = B_2[j : n-1]$
  end if
  if $j = \texttt{length}(B_2)$ **then**
    $S[k : n-1] = B_1[i : n-1]$
  end if
  return S, $cnt_1 + cnt_2 + cnt$
end function

Now since the merge part takes at most n iteration of while loop, we can say the running time for merge is $O(n)$. So, the recursive relation would be:

$$T(n) = 2T(\frac{n}{2}) + O(n) \tag{1}$$

If we use the master theorem for the above recursive relation, we have $T(n) = O(n \log n)$

5. (20 pts.) Problem 5

All recurrences can be solved by unrolling the recurrence and carefully arranging the terms, following the level-by-level approach from lectures (which is equivalent), or using various form of the Master Theorem. However, substitution or induction method is sometimes more convenient to showing the recurrence:

(a) In class, we saw that if $T(n) = a \cdot T(n/b) + O(n^d)$ for some $a > 0, b > 1$ and $d >= 0$, then:

$$T(n) = \begin{cases} O\left(n^d\right) & \text{if } d > \log_b a \\ O\left(n^d \log n\right) & \text{if } d = \log_b a \\ O\left(n^{\log_b a}\right) & \text{if } d < \log_b a \end{cases}$$

Using master Theorem ($a = 9, b = 3, d = 2$), we get $T(n) = O(n^2 \log n)$.

(b) $T(n) = 2T(n/5) + \sqrt{n} = O(\sqrt{n})$ by the Master theorem ($a = 2, b = 5, d = \frac{1}{2}$).

(c) $T(n) = T(n-1) + c^n = \sum_{i=1}^{n} c^i + T(0) = c(\frac{c^n - 1}{c-1}) + T(0) = O(c^n)$ (Formula for the partial sum of a geometric series starting from $i = 1$).

(d) $T(n) = T(n-1) + n^c = \sum_{i=1}^{n} i^c + T(0) = O(n^{c+1})$. (See homework 1 question 4c).

(e) $T(n) = 5T(n/4) + n = O(n^{\log_4 5})$ by the Master theorem.

(f) Unrolling, we have

$$T(n) = 1.5^n + 1.5^{n/2} + 1.5^{n/4} + 1.5^{n/8} + \ldots$$

Then $T(n) \leq \sum_{i=0}^{n} 1.5^i = O(1.5^n)$. Hence, $T(n) = O(1.5^n)$.

(g) $T(n) = 49T(n/25) + n^{3/2} \log n = O(n^{3/2} \log n)$. By unrolling we have:

$$\begin{aligned} \sum_{i=0}^{\log_{25} n} 49^i \left(\frac{n}{25^i}\right)^{3/2} \log\left(\frac{n}{25^i}\right) &= \sum_{i=0}^{\log_{25} n} \left(\frac{49}{25^{3/2}}\right)^i n^{3/2} \log\left(\frac{n}{25^i}\right) \\ &< \sum_{i=0}^{\log_{25} n} \left(\frac{49}{125}\right)^i O(n^{3/2} \log n) \\ &= O(n^{3/2} \log n) \sum_{i=0}^{\log_{25} n} \left(\frac{49}{125}\right)^i \\ &= O(n^{3/2} \log n) O(1) \\ &= O(n^{3/2} \log n) \end{aligned} \tag{2}$$

Note that in the latter equation, we used the fact that the geometric series will be $O(1)$, since $\frac{49}{125} < 1$.

(h) By induction we will show that $T(n) = O(n)$. Assume for $k \leq n-1$, $T(k) \leq ck$, where $c \geq 8$. Then we have:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + T(\frac{n}{8}) + n \leq \frac{7}{8}cn + n \leq cn \tag{3}$$

As a result, we proved that $T(n) \leq 8n$, which means that $T(n) = O(n)$.

(i)
- First method (Unrolling).
  Let us consider the upper bound and suppose $T(n) \leq T(3n/5) + T(2n/5) + cn$ for a suitable constant $c > 0$. Note by unrolling that the size of each sub-problem at level k is of the form $S_{k,i} = \left(\frac{3}{5}\right)^i \left(\frac{2}{5}\right)^{k-i} n$ for some $i = 0, \ldots, k$. Moreover, there are $\binom{k}{i}$ sub-problems of size $S_{k,i}$ at level

k. The height of the recurrence tree is $\log_{5/3} n$. Hence, using the binomial expansion formula:

$$\begin{aligned} T(n) &\leq \sum_{k=0}^{\log_{5/3} n} \sum_{i=0}^{k} \binom{k}{i} c \left(\frac{3}{5}\right)^i c \left(\frac{2}{5}\right)^{k-i} n \\ &= c^2 n \sum_{k=0}^{\log_{5/3} n} \sum_{i=0}^{k} \binom{k}{i} \left(\frac{3}{5}\right)^i \left(\frac{2}{5}\right)^{k-i} \\ &= c^2 n \sum_{k=0}^{\log_{5/3} n} (\frac{3}{5} + \frac{2}{5})^k \\ &= c^2 n \sum_{k=0}^{\log_{5/3} n} 1^k \\ &= O(n \log n) \end{aligned} \tag{4}$$

- Second method (Induction).
  As the induction step, assume that for all $k < n$, we have $T(k) \leq c_1 k \log k$. We want to show that there exists c_1 such that for all large enough n, $T(n) \leq c_1 n \log n$. Since both $\frac{2n}{5}$, and $\frac{3n}{5}$ are less than n according to induction assumption we have: $T(\frac{2n}{5}) \leq c_1 \frac{2n}{5} \log \frac{2n}{5}$, and $T(\frac{3n}{5}) \leq c_1 \frac{3n}{5} \log \frac{3n}{5}$. Now using recurrence relation for $T(n)$ we can write:

$$\begin{aligned} T(n) \leq T(\frac{3n}{5}) + T(\frac{2n}{5}) + cn &\leq c_1 \frac{3n}{5} \log \frac{3n}{5} + c_1 \frac{2n}{5} \log \frac{2n}{5} + cn \\ &= c_1 \frac{3n}{5} \log \frac{3}{5} + c_1 \frac{3n}{5} \log n + c_1 \frac{2n}{5} \log \frac{2}{5} + c_1 \frac{2n}{5} \log n + cn \\ &= c_1 n \log n + c_1 n (\frac{3}{5} \log \frac{3}{5} + \frac{2}{5} \log \frac{2}{5}) + cn \\ &= c_1 n \log n + cn - c_1 n (\frac{3}{5} \log \frac{5}{3} + \frac{2}{5} \log \frac{5}{2}) \end{aligned} \tag{5}$$

  Now, note that if we let $c_1 = \frac{c}{\frac{3}{5} \log \frac{5}{3} + \frac{5}{2} \log \frac{2}{5}}$, and replace it in the latter equation, we get $cn - c_1 n (\frac{3}{5} \log \frac{5}{3} + \frac{5}{2} \log \frac{2}{5}) = 0$, which means that $T(n) \leq c_1 n \log n$. So, induction step is complete, and we proved that $T(n) = O(n \log n)$

(j) Note by unfolding that

$$T(n) = n^{\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^k}} T(n^{\frac{1}{2^k}}) + 10kn.$$

(This can also be proved, for example, by induction.) Now if we let $k = \log_2 \log_2 n$, we have $n^{\frac{1}{2^k}} = 2$ which is constant. Since $\frac{1}{2} \leq \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^k} \leq 1$, we have $T(n) = \Theta(n \log \log n)$.

Rubric:

Problem 1, total points 20

(a) 6 points
3 points: provide correct resulting array for Build-Heap
1 point: provide correct resulting array for the 1st iteration
1 point: provide correct resulting array for the 2nd iteration
1 point: provide correct resulting array for the 3rd iteration

(b) 7 points
4 points: explain the child issue
3 points: explain the child issue is applicable to all the elements in the specified positions

(c) 7 points
3 points: provide a valid counterexample for increasing i from 0 to $\lfloor \frac{n}{2} \rfloor - 1$
4 points: explain why this increasing i from 0 to $\lfloor \frac{n}{2} \rfloor - 1$ fails on this counterexample

Problem 2, 15 pts

- 8 points for proof of time complexity
- 7 points for the justification of why $\Omega(n \log n)$ doesn't apply to the case where M is small and the time is linear time.

Problem 3,15 pts

- 4 points for recurrence $T(n) = T(n/2) + \Theta(1)$, 4 points for running time $\Theta(n)$
- 3.5 points for recurrence $T(n) = T(n/3) + \Theta(1)$, 3.5 points for running time $\Theta(n)$

Problem 4, 20 pts.

The algorithm includes two recursive call, and then a merge and count part.
Designing a correct merge and count procedure: 5
The correctness of algorithm: 5
Deriving the recurrence relation: 5
Showing the run-time from the recurrence relation: 5

Problem 5, 30 pts

There are total of 10 parts, each part is worth 3 pts. For the parts that can be solve by master theorem, only correct answer get the full points, and there is not partial credit for these parts. For, the parts that master theorem can not be applied, unrolling correctly is worth 1.5 pts, and showing the final answer is worth 1.5 pts.

CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 4

1. (20 pts.) Problem 1

(a) Similar to binary search, start by examining $A[\lfloor \frac{n}{2} \rfloor]$.

- If $A[\lfloor \frac{n}{2} \rfloor]$ is $\lfloor \frac{n}{2} \rfloor$, then we have a satisfactory index.
- If $A[\lfloor \frac{n}{2} \rfloor] > \lfloor \frac{n}{2} \rfloor$, then no element in the second half of the array can possibly satisfy the condition. To check this, note that all integers in the array are distinct, so each integer in the array is at least one greater than the previous element. Since $A[\lfloor \frac{n}{2} \rfloor]$ is already greater than $\lfloor \frac{n}{2} \rfloor$, all the elements on its right are greater than their indices as well. Thus, the second half can be discarded in this case.
- if $A[\lfloor \frac{n}{2} \rfloor] < \lfloor \frac{n}{2} \rfloor$, then by the same logic no element in the first half of the array can satisfy the condition.

We discard the half of the array that cannot hold an answer and repeat the same check until the satisfactory index has been found or all the elements in the array have been discarded.

(b) At each step we do a single comparison and discard at least half of the remaining array (or terminate), so the running time of this algorithms is given by the recurrence $T(n) = T(n/2) + O(1)$ and hence $T(n) = O(\log n)$ time by the master theorem.

2. (20 pts.) Problem 2

(a)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^2 = \begin{bmatrix} a^2+bc & ab+bd \\ ca+dc & cb+d^2 \end{bmatrix} = \begin{bmatrix} a^2+bc & b(a+d) \\ c(a+d) & bc+d^2 \end{bmatrix}$$

Hence, the 5 multiplications $a^2, d^2, bc, b(a+d)$ and $c(a+d)$ suffice to compute the square.

(b) Two possible answers:

1) We have:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^2 = \begin{bmatrix} A^2+BC & AB+BD \\ CA+DC & CB+D^2 \end{bmatrix} \neq \begin{bmatrix} A^2+BC & B(A+D) \\ C(A+D) & BC+D^2 \end{bmatrix}.$$

Note that matrices don't commute! This means $BC \neq CB$ in general, so we cannot reuse that computation. Also, matrix multiplication only has left-distributivity and right-distributivity. As a result, $B(A+D) = BA+BD \neq AB+BD$ and $C(A+D) = CA+CD \neq CA+DC$. Thus, we end up getting more than 5 subproblems, and the recurrence $T(n) = 5T(n/2) + O(n^2)$ does not make sense.

2) Originally, our problem is to square a matrix. However, to compute $\begin{bmatrix} A & B \\ C & D \end{bmatrix}^2$, there is no way of making all the subproblems the same as our initial problem (squaring a matrix). So, this can't be claimed as a divide-and-conquer approach.

(c) Given two $n \times n$ matrices X and Y, create the $2n \times 2n$ matrix A:

$$A = \begin{bmatrix} 0 & X \\ Y & 0 \end{bmatrix}$$

Now, it suffices to compute A^2 to obtain the result of the matrix multiplication, as its upper left block will contain XY:

$$A^2 = \begin{bmatrix} XY & 0 \\ 0 & YX \end{bmatrix}$$

Let $S(n)$ be the time to square a $n \times n$ matrix. The general matrix multiplication XY can be calculated in time $S(2n)$. Given that $S(n) = O(n^c)$, $S(2n) = O((2n)^c) = O(2^c n^c) = O(n^c)$ since c is a constant as well as 2^c which can be ignored as a coefficient.

Note: matrix A in the solution is not unique. The other possibilities for matrix A are $\begin{bmatrix} 0 & X \\ 0 & Y \end{bmatrix}$ ($A^2 = \begin{bmatrix} 0 & XY \\ 0 & Y^2 \end{bmatrix}$), $\begin{bmatrix} X & Y \\ 0 & 0 \end{bmatrix}$ ($A^2 = \begin{bmatrix} X^2 & XY \\ 0 & 0 \end{bmatrix}$), $\begin{bmatrix} Y & 0 \\ X & 0 \end{bmatrix}$ ($A^2 = \begin{bmatrix} Y^2 & 0 \\ XY & 0 \end{bmatrix}$), $\begin{bmatrix} 0 & 0 \\ Y & X \end{bmatrix}$ ($A^2 = \begin{bmatrix} 0 & 0 \\ XY & X^2 \end{bmatrix}$), and $\begin{bmatrix} 0 & Y \\ X & 0 \end{bmatrix}$ ($A^2 = \begin{bmatrix} YX & 0 \\ 0 & XY \end{bmatrix}$).

3. (20 pts.) Problem 3

It is sufficient to show how to find n in time $O(\log n)$, as we can use binary search on the array $A[0, \cdots, n-1]$ to find any element x in time $O(\log n)$. To find n, query elements $A[0], A[1], A[2], A[4], A[8], \cdots, A[2^i], \cdots$,until you find the first element $A[2^k]$ such that $A[2^k] = \infty$. Then, $2^{k-1} \leq n < 2^k < 2n$. We can then do binary search on $A[2^{k-1}, \cdots, 2^k]$ to find the last non-infinite element of A. This takes time $O(\log(2^k - 2^{k-1})) = O(\log n)$.

4. (20 pts.) Problem 4

We will assume $k \leq n$. If this is not the case, then for $k' = 2n + 1 - k$, we are trying to find the k'-th largest element, and $k' \leq n$. But finding the k-th largest element and finding the k-th smallest element are symmetric problems, so with some minor tweaks we can use the same algorithm. So for brevity, we will only give the answer for when $k \leq n$.

Since $k \leq n$, the k-th smallest element can't exist past index k of either array. So we cut off all but the first k elements of both arrays. Now, we just want to find the median of the union of the two arrays. To do this, we check the middle elements of both arrays, and compare them. If, the median of the first list is smaller than the median of the second list, then the median can't be in the left half of the first list or the right half of the second list. If the median of the second list is smaller, the opposite is true. So we can cut these elements off, and then recurse on the resulting arrays.

Pseudocode

procedure TWOARRAYSELECTION($a[1..n]$, $b[1..n]$, element rank k)
  Set $a := a[1, \cdots, k]$, $b := b[1, \cdots, k]$,
  Let $\ell_1 = \lfloor k/2 \rfloor$ and $\ell_2 = \lceil k/2 \rceil$
  while $a[\ell_1] \neq b[\ell_2]$ and $k > 1$ **do**
    if $a[\ell_1] > b[\ell_2]$ **then**
      Set $a := a[1, \cdots, \ell_1]$; $b := b[\ell_2 + 1, \cdots, k]$; k := ℓ_1
      Let $\ell_1 = \lfloor k/2 \rfloor$ and $\ell_2 = \lceil k/2 \rceil$
    else
      Set $a := a[\ell_1 + 1, \cdots, k]$; $b := b[1, \cdots, \ell_2]$; $k := \ell_2$
      Let $\ell_1 = \lfloor k/2 \rfloor$ and $\ell_2 = \lceil k/2 \rceil$

```
        end if
    end while
    if k = 1 then
        return min(a[1], b[1])
    else
        return a[ℓ1]
    end if
end procedure
```

Proof of correctness

Our algorithm starts off by comparing elements $a[\ell_1]$ and $b[\ell_2]$. Suppose $a[\ell_1] > b[\ell_2]$.
Then, in the union of a and b there can be at most $k-2$ elements smaller than $b[\ell_2]$, i.e. $a[1,\cdots,\ell_1-1]$ and $b[1,\cdots,\ell_2-1]$, and we must necessarily have $s_k > b[\ell_2]$. Similarly, all elements $a[1,\cdots,\ell_1]$ and $b[1,\cdots,\ell_2]$ will be smaller than $a[\ell_1+1]$; but these are k elements, so we must have $s_k < a[\ell_1+1]$.
This shows that s_k must be contained in the union of the subarrays $a[1,\cdots,\ell_1]$ and $b[\ell_2+1,\cdots,k]$. In particular, because we discarded ℓ_2 elements smaller than s_k, s_k will be the ℓ_1th smallest element in this union.
We can then find s_k by recursing on this smaller problem. The case for $a[\ell_1] < b[\ell_2]$ is symmetric.
If we reach $k = 1$ before $a[\ell_1] = b[\ell_2]$, we can cut the recursion a little short and just return the minimum element between a and b. You can make the algorithm work without this check, but it might get clunkier to think about the base cases.

Alternatively, if we reach a point where $a[\ell_1] = b[\ell_2]$, then there are exactly k greater elements, so we have $s_k = a[\ell_1] = b[\ell_2]$.

Running time analysis

At every step we halve the number of elements we consider, so the algorithm will terminate in $\log(k)$ recursive calls. Assuming the comparison takes constant time, the algorithm runs in time $\Theta(\log k)$. Note that this does not depend on n.

5. (20 pts.) Problem 5

The intuition for the divide-and-conquer solution is to split the vector v into a top half v_T with the first $\lfloor n/2 \rfloor$ elements and a bottom half v_B with the rest of the elements. We'll similarly split the result of the multiplication $(H_k v)_T$ and $(H_k v)_B$. Using this notation and the recursive nature of H_k, we can split $H_k v$ as follows:

$$H_k v = \begin{bmatrix} (H_k v)_T \\ (H_k v)_B \end{bmatrix} = \begin{bmatrix} H_{k-1} v_T + H_{k-1} v_B \\ H_{k-1} v_T - H_{k-1} v_B \end{bmatrix} = \begin{bmatrix} H_{k-1}(v_T + v_B) \\ H_{k-1}(v_T - v_B) \end{bmatrix}$$

This shows that we can find $H_k v$ by calculating $v_T + v_B$ and $v_T - v_B$ and using this algorithm to recursively multiply $H_{k-1}(v_T + v_B)$ and $H_{k-1}(v_T - v_B)$. The running time of this algorithm is given by the recurrence relation $T(n) = 2T(\frac{n}{2}) + O(n)$, where the linear term is the time taken to perform the two sums. This has solution $T(n) = O(n \log n)$ by the Master theorem.

Rubric:

Problem 1, 20 pts

(a) 15 points
6 points: provide an algorithm that achieves what we want
4 points: it's truly faster than the brute force algorithm
5 points: explain why it's correct

(b) 5 points
3 points: provide a running time analysis that makes sense
2 points: reach to a correct conclusion on the running time
Note:

- Some students might analyze the running time correctly, but the given algorithm is not faster than the brute force way. In this case, they'll get full points for part (b) but lose 4 points for part (a).
- Some other students might provide an algorithm that's indeed faster than the brute force approach, but their running time analysis has certain issues. In this case, they'll have the 4 points for part (a) second rubric but lose points for part (b).

Problem 2, 20 pts

(a) 5 points
3 points: correctly expand the squaring of a 2×2 matrix
2 points: correctly identify the 5 needed multiplications

(b) 6 points as long as the student provides either of the possible answers

1) 3 points: correctly expand the squaring of a matrix with 4 sub-matrices and point out the non-commutativity property
   3 points: explain how the non-commutativity property introduces an issue for the statement
2) 3 points: point out the sub-problems of squaring a matrix are not necessarily squaring a matrix
   3 points: point out it's no longer divide-and-conquer if the main problem and its sub-problems are different

(c) 9 points
3 points: the explanation of squaring a $2n \times 2n$ matrix can also be done in $O(n^c)$ time makes sense
2 points: specify a reasonable $2n \times 2n$ matrix A
4 points: explain how they can obtain the result of a general matrix multiplication from A^2

Problem 3, 20 pts

(a) 5 pts for pointing out that to find any element in a sorted array, the running time is $O(\log n)$

(b) 15 pts for solution how to find the last non-infinite element in the array

Problem 4, 20 pts

(a) 8 pts for designing correct divide and conquer algorithm, and clear description of their algorithm. Note that correct algorithm should give us the k^{th} element in $O(\log k)$, however, if their runtime was $O(\log n)$, they will still get $7 pts$ for this part.

(b) 6 pts for showing the correctness of their algorithm. They need to show that why their algorithm outputs k_{th} element

(c) 6 pts for analyzing the running time. Two ways of doing this are: 1) either showing their algorithm has $\log k$ recursive calls, and each call takes $O(1)$, so the total runtime will be $O(\log k)$. 2) or they can write the recurrent relation ($T(n) = T(\frac{n}{2}) + O(1)$), and derive runtime from it. In this case writing recurrence gets 4 pts, and deriving the final runtime gets 2 pts.

Problem 5, 20 pts

(a) 5 pts for identifying that the problem can be split into top and bottom

(b) 10 pts for splitting the multiplication into 2 multiplications of size $n/2$

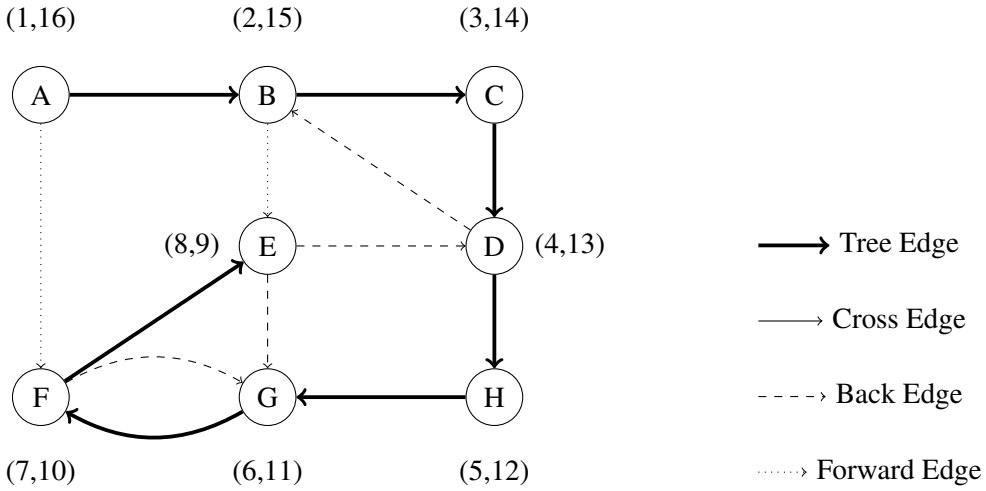
(c) 5 pts for showing the running time is $O(n \log n)$

CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

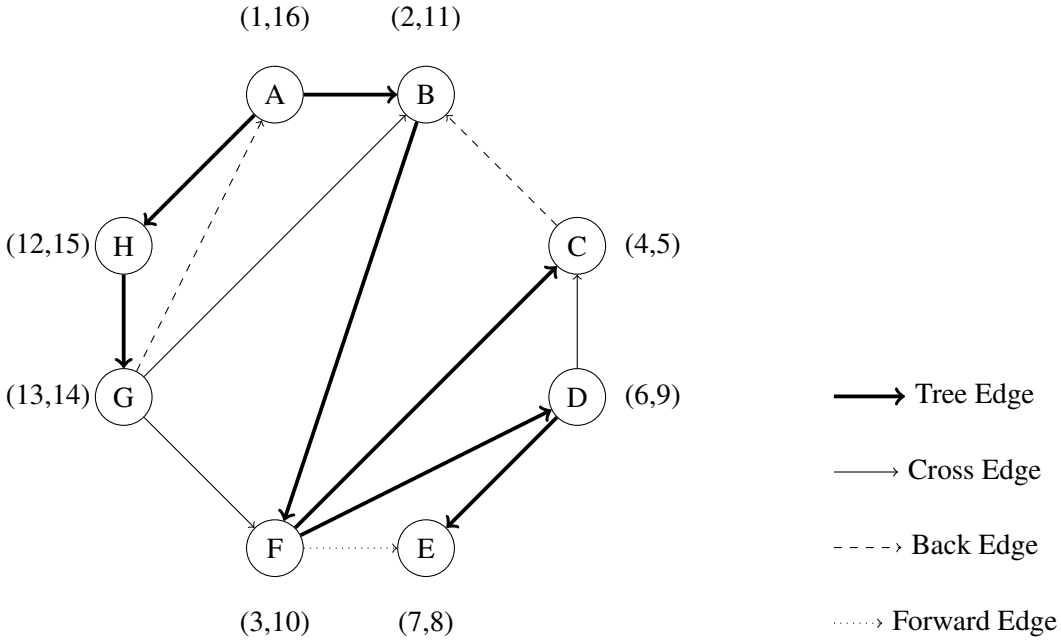
HW 5

1. (21 pts.) Problem 1

(a)



(b)



2. (21 pts.) Problem 2

(a) $A : 1, 14$
$B : 15, 16$
$C : 2, 13$
$D : 3, 10$
$E : 11, 12$
$F : 4, 9$
$G : 5, 6$
$H : 7, 8$

(b) Sources: A, B; Sinks: G, H

(c) B, A, C, E, D, F, H, G

(d) Any ordering of the graph must be of the form $\{A,B\}, C, \{D,E\}, F, \{G,H\}$, where $\{A,B\}$ indicates A and B may be in any order within these two places. So, the total number of orderings is $2^3 = 8$.

3. (20 pts.) Problem 3

(a) We create a graph $G = (V,E)$ where vertices represent locations and edges represent roads. Because the roads are two-way, the graph must be undirected. In this graph, locations that can be driven between are represented by connected components. The graph might have several connected components, separated by the "rivers." An edge (u,v) in this graph is safe to remove if it does not separate its connected component. Because the edge connects u to v, u and v must start in the same connected component. If (u,v) is removed and v is still reachable from u, then the connected component must have contained a cycle that included (u,v). This logic is the reverse of worksheet 5 problem 4. So there is some road that is safe to close if and only if our graph G has a cycle.

(b) We can't just use DFS to check for cycles, since that takes time $O(|V|+|E|)$. However, we can modify DFS to terminate the first time we see an edge that goes back to a visited vertex; this is a minor modification to the Explore procedure. This algorithm will detect if there is a cycle.

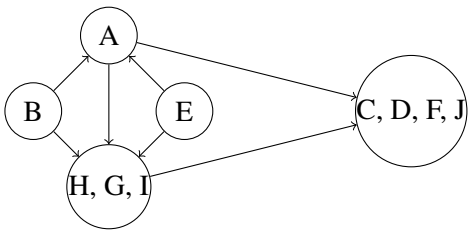
Let us analyze the running time of this modified DFS algorithm. The key observation is that a graph with more than $|V|$ edges will always have a cycle. (If you don't see why, you should prove this fact. Essentially, if $|E| \geq |V|$ you are forced to create a cycle.) Suppose first that the graph has no cycles. Then, $|E| < |V| - 1$. So, in this case, the running time of $O(|E|+|V|)$ of DFS is actually $O(|V|)$. Now, suppose that the algorithm stopped early. This is because it found some edge coming from the currently considered vertex to a vertex that has already been considered. Since all of the edges considered up to this point didn't do that, we know that they formed a forest. So, the number of edges considered is at most the number of vertices considered, which is $O(|V|)$. Consequently, the total running time is $O(|V|)$.

4. (20 pts.) Problem 4

(i) (a) The strongly connected components are found in the order $\{C,D,F,J\}$, $\{G,H,I\}$, $\{A\}$, $\{E\}$, $\{B\}$.

(b) The source SCC's are $\{E\}$ and $\{B\}$, while $\{C,D,F,J\}$ is a sink SCC.

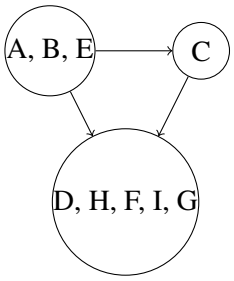
(c)



(d) It is necessary to add two edges to make the graph strongly connected, e.g. by adding $C \to B$ and $B \to E$.

(ii) (a) The strongly connected components are found in the order $\{D,F,G,H,I\}$, $\{C\}$, $\{A,B,E\}$.

(b) $\{A,B,E\}$ is a source SCC, while $\{D,F,G,H,I\}$ is a sink.



(c)

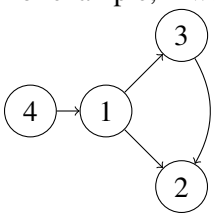
(d) In this case, adding one edge from any vertex in the sink SCC to any vertex in the source SCC makes the metagraph strongly connected and hence the given graph also becomes strongly connected.

5. (19 pts.) Problem 5 **Solution**

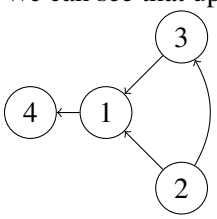
(a) We construct the directed graph $G = (V,E)$, where every node represents a student (numbered 1 to n), and every edge (u,v) represents u being friends with v. If v is also friends with u, another edge (v,u) will also be present. In this graph, the first-pick partner of student s is the node of minimum value reachable from node s.

(b) Here, if we reverse the direction of every edge in the graph, the set of students who have good student k in their social network (k is reachable), becomes precisely the set of students that good student k can reach. Furthermore, all vertices within the same SCC should ultimately receive the same label (the best student in that SCC), along with all the vertices within any other SCC that our current SCC can reach. Thus, for all unvisited nodes in the reverse graph, we can run DFS starting from the lowest-ranked unvisited node i, and for all the nodes j that are reachable from i, we assign i to be student j's first-pick partner.

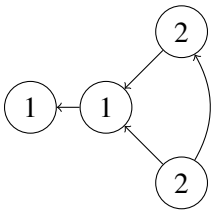
For example, if we start with the graph:



We can see that upon reversing it we'd get the graph:



When we run DFS on this graph, we'll need two 'starts'. Once starting at 1 (the best-ranked student) and then again starting at 2 (the best-ranked student we haven't visited yet) after we hit a dead end. Our two runs produce the following labels:



Looking back at our original graph, we can see that this does indeed correspond to the right answer!

We can see that this algorithm is very similar to our algorithm for finding SCCs. Rather than starting from a vertex in a sink SCC, however, we start from the vertex with the smallest value. So rather than finding SCCs one at a time, we'll first find all the vertices in the SCCs that can reach 1, then all the vertices in SCCs that can reach the next smallest available number, and so on.

Algorithm

Let G^R be the graph G with its edge directions reversed. The algorithm is as follows.

Function `Choosing-Partners(`G`)`**:**

```
while there are unvisited nodes in G^R do
    Run DFS on G^R starting from the numerically-first unvisited node i
    for j visited by this DFS do
        first_pick[j] := i
    end
end
```

To see that this algorithm is correct, note that if a vertex i is assigned a value then that value is the smallest of the nodes that can reach it in G^R, and every node is assigned a value because the loop does not terminate until this happens. Now observe that the set of vertices reachable by i in G^R is the set of vertices which can reach i in G.

The running time is $O(|V|+|E|)$ since computing G^R can be done in linear time, and we process every vertex and edge exactly once in the DFS.

Rubric:

Problem 1, 21 pts

(a)
- 0.5 pts for every node's pre number and post number
- 0.5 pts for every edge type

(b)
- 0.5 pts for every node's pre number and post number
- 0.5 pts for every edge type

Problem 2, 20 pts

(a) 8 points: 0.5 point for a correct pre number and a correct post number, respectively, for each vertex.

(b) 2 points: 0.5 point for each correctly categorized vertex.

(c) 4 points: 0.5 point for each correctly ordered vertex.

(d) 6 points
1 point: find a pair of nodes that can be swapped in other correct topological ordering
2 points: provide the correct final answer.
1 point: provide an explanation on how those pairs of nodes lead to the correct answer.

Problem 3, 20 pts

(a)
- 2 pts: Undirected graph where nodes represent locations and edges represent roads.
- 3 pts: States that "safe-to-close roads" implies finding cycles.

(b)
- 5 pts: Relating the back edge found by DFS to existence of a cycle in undirected graph.
- 5 pts: modifying the Explore procedure such that it finds a back edge, and terminates early (just an explanation would be enough, they do not need to write a pseudocode)
- 5 pts: analyzing the runtime of the algorithm, and explain why it would be independent of $|E|$ if we terminate the algorithm early

Problem 4, 20 pts

(a) 2 points for (i) and (ii), respectively

(b) 2 points for (i) and (ii), respectively

(c) 4 points for (i) and (ii), respectively

(d) 2 points for (i) and (ii), respectively

Problem 5, 19 pts

(a)
- 2 pts for directed graph
- 2 pts for the right description of node
- 2 pts for the right description of edge
- 3 pts for the questions raised by this problem

(b)
- 5 pts for reverse graph and some indication of why it's helpful
- 5 pts for exploring the reverse graph from the next best-ranked student (node), marking all reachable nodes with that student's number.

CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 6

1. (20 pts.) Problem 1

Run DFS on the tree starting from r and store the previsit and postvisit times for each node. Since the given graph is a tree, and we started at the root, the DFS tree is the same as the given tree. We would like to answer queries of the form "is u an ancestor of v?" in $O(1)$ time. For this, note that u is an ancestor of v in the DFS tree if and only if $pre(u) < pre(v) < post(v) < post(u)$. After computing all the pre and post numbers, this condition can be checked for any pair of vertices u and v in $O(1)$ time.

2. (20 pts.) Problem 2

(a) This setting can be represented as a directed graph. We view the intersections as its vertices with the streets being directed edges, since they are one-way. Then the claim is equivalent to saying that this graph is strongly connected. This is true if and only if the graph has only one SCC, which can be checked by running the SCC algorithm in linear time.

(b) The weaker claim says that starting from the town hall, one cannot get to any other SCC in the graph. This is equivalent to saying that the SCC containing the vertex corresponding to the town hall is a sink component. Regarding how the claim can be checked in linear time, there are two possible answers:

(1) First find all the SCCs by SCC algorithm, and then running Explore from the vertex corresponding to the town hall (in the original graph). If any vertices visited through this Explore procedure are not in the SCC that the town hall belongs to, then this SCC is not a sink SCC. Otherwise, it is. The running time is linear because the running time of both the SCC algorithm and Explore is linear, as we know from class.

(2) Run the SCC algorithm. Note that this algorithm for finding SCCs progressively discards sinks SCCs one by one. A component found by the algorithm is a sink if and only if there are no edges going out of the component. We use this to check if the SCC containing the town hall vertex is a sink component. The running time is linear because the running time of the SCC algorithm is linear.

3. (20 pts.) Problem 3

We start by linearizing (topological sorting) G, which we know is possible because it is a DAG. Any path from c to s can only pass through vertices between c and s in the topological order and hence we can ignore the other vertices.

Let $c = v_0, v_1, \ldots, v_k = s$ be the vertices from c to s in the topological order. For each i, we count the number of paths from c to v_i as n_i. Each path to a vertex i and an edge (i, j), gives a path the vertex j and hence

$$n_j = \sum_{(i,j)\in E} n_i$$

Since $i < j$ for all $(i, j) \in E$, this can be computed in increasing order of j. The required answer is n_k. Finding the topological order requires running DFS, which takes $O(|V| + |E|)$ time. Then, computing the sum considers every vertex between c and s in the topological sort, or at most $O(|V|)$ vertices. Specifically, it must retrieve the sum from all of that vertex's already-computed neighbors, so at worst it considers $O(|E|)$ edges total across every sum. Therefore the final running time is $O(|V| + |E|)$.

4. (20 pts.) Problem 4

(a) Assume it is possible to have BFS forward edges. Assume one of this type of edges is (u,w), which leads a node u to its non-child descendant w. As u's descendant, w is visited after u is visited. Due to the edge (u,w), $dist[w] = dist[u] + 1$. So, w is u's child which contradicts with the earlier assumption that w is u's non-child descendant. Therefore, it is impossible to have BFS forward edges.

(b) Algorithm:

- Run BFS on G and obtain the BFS tree. The edges in the BFS tree are tree edges.
- Intuition: both BFS back edges and DFS back edges require the tail to lead to its ancestor as the head. Also, both BFS cross edges and DFS cross edges require the tail to lead to a node that's neither its ancestor nor its descendant as the head. The only difference between BFS edges and DFS edges are that one is regarding the BFS tree and the other is regarding the DFS tree. So, the key to classify BFS back edges and BFS cross edges is to determine the ancestor/descendant relationship between the tail and the head of the edges. We can make use of pre-visit number and post-visit number of each node in a tree to achieve this, and those numbers can be obtained by running DFS on the tree.
- Run DFS on BFS tree to obtain the pre-visit number and post-visit number for each vertex on the BFS tree.
- To classify the edges that are not in the BFS tree (non tree edges), for an edge (w,v), if $pre[v] < pre[w] < post[w] < post[v]$, then it's a back edge. If $pre[v] < post[v] < pre[w] < post[w]$ or $pre[w] < post[w] < pre[v] < post[v]$, then it's a cross edge.

Run time analysis:

- BFS takes $O(|V| + |E|)$ time.
- DFS takes $O(|V| + |E|)$ time.
- Going through the edge set takes $O(|E|)$ time.

Thus, overall, the algorithm takes linear time, $O(|V| + |E|)$.

5. (20 pts.) Problem 5

(a) Notice that in order to have a directed edge from vertex u to vertex v, the clause $\hat{u} \vee v$ needs to be in ϕ. Hence, the edge from u to v is equivalent to $u \Rightarrow v$. Consequently, a path from x to y means that $x \Rightarrow y$. If x and $\hat{x}$ are in the same strongly connected component, then we have $x \Rightarrow \hat{x}$ and $\hat{x} \Rightarrow x$. Then there is no way to assign a value to x to satisfy both these implications.

(b) **Consider the following algorithm:** We recursively find sinks in the graph of strongly connected components and assign true to all the literals in the sink component (this means that if a sink component contains the literal $\hat{x}$, then we assign $\hat{x} = \texttt{true}$ and $x = \texttt{false}$). We will then remove all the variables which have been assigned a value from the graph.

Correctness: The algorithm will output an assignment. Assume on the contrary that the assignment does not satisfy all the clauses, say, a clause $\alpha \vee \beta$ is not satisfied by the assignment, where α and β are literals. So the algorithm sets $\alpha = \texttt{false}$, $\beta = \texttt{false}$, $\hat{\alpha} = \texttt{true}$, $\hat{\beta} = \texttt{true}$. Note that the algorithm sets α and $\hat{\alpha}$ at the same time, and similarly for β and $\hat{\beta}$.

- Case 1 - the algorithm sets α before β: at the time the algorithm sets $\hat{\alpha} = \texttt{true}$, $\hat{\alpha}$ is in a sink component. Since β and $\hat{\beta}$ are set at a later time, the vertices for β and $\hat{\beta}$ must still exist. So,

the edge $\hat{\alpha} \Rightarrow \beta$ is still in the graph just before $\hat{\alpha}$ is removed, and hence β must also be in the same sink component. So the algorithm should set $\beta = \texttt{true}$ at the same time it sets $\hat{\alpha} = \texttt{true}$, a contradiction.

- Case 2 - the algorithm sets α and β at the same time: the algorithm sets $\hat{\alpha} = \texttt{true}$ and $\hat{\beta} = \texttt{true}$ at the same time, so they are in the same sink component. Right before they are set, the edge $\hat{\alpha} \Rightarrow \beta$ is still in the graph, so β is in the same component as $\hat{\alpha}$. So β is in the same sink component as $\hat{\beta}$, implying that they are in the same strongly connected component (at the time they are set, and hence in the original graph), a contradiction.
- Case 3 - the algorithm sets α after β: symmetric to Case 1, a contradiction.

(c) To perform the operations in the previous part, we only need to construct this graph from the formula, find its strongly connected components, and identify the sinks - all of which can be done in linear time.

Rubric:

Problem 1, 20 pts

- 10 pts for a DFS from root r
- 10 pts for u is an ancestor of v if and only if $pre(u) < pre(v) < post(v) < post(u)$

Problem 2, 20 pts

(a) 10 points
4 points: correctly formulate the setting as a directed graph.
3 points: correctly state what the mayor claims in graph theory terminology.
3 points: provide an algorithm that can check the claim and explain why the algorithm takes linear time.

(b) 10 points
5 points: correctly state what the weaker claim says in graph theory terminology.
5 points: provide an algorithm that can check this weaker claim and explain why the algorithm takes linear time.

Problem 3, 20 pts

- 10 pts - topological sort
- 5 pts - only consider vertices between the given c and s
- 5 pts - correct sum of path counts

Problem 4, 20 pts

(a) 6 points: provide an explanation, and it makes sense.

(b) 14 points
10 points: provide an algorithm that can correctly categorize the edges in linear time.
4 points: explain why this algorithm takes linear time.

Problem 5, 20 pts

(a) 8pts
- 2pts for if there is an edge from u to v, there should a clause $\hat{u} \vee v$ in ϕ
- 2pts for the path from x to y means $x \Rightarrow y$
- 4pts for conclusion why x and $\hat{x}$ can not be in the same SCC

Or any reasonable explanation worth 8 pts

(b) 8pts - reasonable explanation of why the given algorithm is correct.

(c) 4pts
- 1pts for construct the graph
- 1pts for find the connected components
- 1pts for identify the sinks
- 1pts make the conclusion of linear time

CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 7

1. (20 pts.) **Dijkstra's Algorithm.**

Solution:

(a) The order of vertices in which they are removed from the priority queue is s, a, c, d, b.

(b) As the order of vertices being removed from the priority queue needs to be same as before, we need to guarantee $distance(s,c) < distance(s,d) < distance(s,b)$. Changing $l(a,d)$ only potentially affects $distance(s,d)$ and is not possible to affect $distance(s,c)$ as well as $distance(s,b)$. So, we are certain that $distance(s,c) = 3$ and $distance(s,b) = 7$. Thus, our goal is to make sure $3 < distance(s,d) < 7$.

$$distance(s,d) = \begin{cases} l(s,a) + l(a,d) = 1 + l(a,d) & \text{if } 1 + l(a,d) \leq 6 \\ l(s,a) + l(a,c) + l(c,d) = 6 & \text{if } 6 \leq 1 + l(a,d) \end{cases}$$

With the consideration of $3 < distance(s,d) < 7$, the condition of the first case ($(s,a) \to (a,d)$ is the shortest path from s to d) is $3 < 1 + l(a,d) \leq 6$, and the the condition of the second case ($(s,a) \to (a,c) \to (c,d)$ is the shortest path from s to d) is still $6 \leq 1 + l(a,d)$. Overall, as we consider both cases, the range of $l(a,d)$ is $(2, \infty)$.

2. (20 pts.) **Unique Shortest Path.**

Solution: This can be done by slightly modifying Dijkstra's algorithm. The array `usp[·]` is initialized to `true` in the initialization loop. The main loop is modified as follows, with the central change being an extra equality check between dist(v) and dist(u)+$l(u,v)$.

```
while H is not empty do
  u = GetMin()
  Remove(H, 0)
  for all (u,v) ∈ E do
    if dist(v) = dist(u) + l(u,v) then
      usp[v] = false
    end if
    if dist(v) > dist(u) + l(u,v) then
      dist(v) = dist(u) + l(u,v)
      usp[v] = usp[u]
      DecreaseKey(H, position[v], dist[v])
    end if
  end for
end while
```

This algorithm works because, for any node u that is removed from the heap, no new shortest paths to u can be discovered. This is due to the edge weights all being positive. Then, for every edge (u,v), if going through u represents a new shortest path to v, we know that the shortest path to v can be unique only if the shortest path to u is also unique. Finally, if a path to v is uncovered that has equal length to the existing shortest path to v, we know there are at least two shortest paths to v. This could be reset by later discovering a shorter path to v in the future.

3. (20 pts.) **Start Node Negative Edges.**

Solution: Dijkstra's algorithm would work.

Proof: Consider the proof of Dijkstra's algorithm. The proof depended on the fact that if we know the shortest paths for a subset $S \subseteq V$ of vertices, and if (u,v) is an edge going out of S such that v has the minimum estimate of distance from s among the vertices in $V \backslash S$, then the shortest path to v consists of the (known) path to u and the edge (u,v). We can argue that this still holds even if the edges going out of the vertex s are allowed to be negative. Let (u,v) be the edge out of S as described above. For the sake of contradiction, assume that the path claimed above is not the shortest path to v. Then there must be some other path from s to v which is shorter. Since $s \in S$ and $v \notin S$, there must be some edge (i,j) in this path such that $i \in S$ and $j \notin S$. But then, the distance from s to j along this path must be greater than the estimate of v, since v had the minimum estimate. Also, the edges on the path between j and v must all have non-negative weights since the only negative edges are the ones out of s. Hence, the distance along this path from s to v must be greater than the estimate of v, which leads to a contradiction.

4. (20 pts.) **Shortest Path in Currency Trading.**

Solution:

a) Represent the currencies as the vertex set V of a complete directed graph G. To find the most advantageous ways to converts c_s into c_t, you need to find the path $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ maximizing the product $r_{i_1,i_2} r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k}$. This is equivalent to minimizing the sum $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}})$. Hence, it is sufficient to find a shortest path in the graph G with weights $w_{ij} = -\log r_{ij}$. Because these weights can be negative, we apply the Bellman-Ford algorithm for shortest paths to the graph , taking s as origin.

b) Just iterate the updating procedure once more after $|E||V|$ rounds. If any distance is updated, a negative cycle is guaranteed to exist, i.e. a cycle with $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}}) < 0$, which implies $\prod_{j=1}^{k-1} r_{i_j,i_{j+1}} > 1$, as required.

5. (20 pts.) **Faster All-Pairs Shortest Path.**

(a) The total weight of a path is the sum of the weights of its edges. When we expand the reweighting formula, the intermediate h values cancel each other out. For example, edge (v_2,v_3) has $-h(v_3)$, but the next edge (v_3,v_4) has $+h(v_3)$. The only h values missing a pair to cancel with are $+h(v_0)$ and $-h(v_k)$:

$$\sum_{i=0}^{k-1} \hat{w}(v_i,v_{i+1}) = \sum_{i=0}^{k-1} (w(v_i,v_{i+1}) + h(v_i) - h(v_{i+1})) = h(v_0) - h(v_k) + \sum_{i=0}^{k-1} w(v_i,v_{i+1})$$

We are left with the sum of the original weights, which is the total weight of the original path, plus $h(v_0) - h(v_k)$. Note that any path from v_0 to v_k shares those endpoints, so $h(v_0) - h(v_k)$ is constant and the total weights of every path maintain their relative ordering. Therefore, the original shortest path is still the shortest path after reweighting.

(b) This part is similar to (a). The total weight of a cycle is the sum of the weights of its edges. The only difference is that here, the starting and ending points are equal, so $h(v_0) = h(v_k)$. This means that every h value will cancel.

$$\sum_{i=0}^{k-1} \hat{w}(v_i,v_{i+1}) = \sum_{i=0}^{k-1} (w(v_i,v_{i+1}) + h(v_i) - h(v_{i+1})) = \sum_{i=0}^{k-1} w(v_i,v_{i+1})$$

Reweighting preserves the exact total weight of every cycle. Therefore, negative reweighted cycles can occur if and only if the original cycle was also negative.

(c) The central observation is that because h is defined using shortest paths from s, the shortest path from s to v must at most equal in length to the shortest path from s to u plus the distance between u and v. This is true because either the path through u is the shortest path to v, in which case they are equal, or the path through u is not the shortest path to v, in which case the shortest path to v must be even shorter. This inequality can be formalized as $h(v) \leq h(u) + w(u,v)$. Subtracting $h(v)$ gives $w(u,v) + h(u) - h(v) \geq 0$. This is exactly our formula for $\hat{w}(u,v)$, so $\hat{w}(u,v) \geq 0$.

(d) We are comparing Floyd-Warshall's $O(|V|^3)$ to this algorithm's $O(|V|^2 \log |V| + |V||E|)$. $|V|^2 \log |V|$ is always preferable to $|V|^3$; the problem is $|V||E|$. In a graph with $O(|V|^2)$ edges (such as a complete graph), $O(|V||E|) = O(|V|^3)$. So, the condition for this algorithm to be faster is that the number of edges must not be $\Omega(|V|^2)$. For example, in a tree, there are $O(|V|)$ edges and Johnson's algorithm is dominated by $O(|V|^2 \log |V|)$, which is much better than $O(|V|^3)$. In short, we say that Johnson's Algorithm is preferable for sparse graphs.

6. (10 pts.) **Extra Credit.**

Solution: We will modify Dijkstra's algorithm to solve this problem. Dijkstra keeps a set A, and $dist(u)$ for all $u \notin A$. A contains nodes whose shortest path from s are already computed. For $u \notin A$, $dist(u)$ stores the shortest $s \to u$ path among the paths that start at s, and use only nodes in A before ending at u. At each iteration, Dijkstra adds the u with the smallest $dist(u)$ to A, and update the $dist(v)$ for all v where $(u,v) \in E$.

In particular, we will change the order of how the nodes are added to A. We first recognize the SCC of our graph treating roads as bi-directional edges. If two nodes are connected by roads, then they must be in the same SCC, and the additional constraint about planes ensures that no plane edge will be inside a SCC. Thus the edges across different SCCs are exactly the plane routes.

We first find the topological order of the metagraph of SCCs, and for each node u in an SCC, let $r(u)$ be the index of the SCC in the topological ordering. It is easy to see that if $r(u) = r(v)$, then any path from u to v must be all roads, and if $r(u) > r(v)$, there won't be any path from u to v.

The modification we make to Dijkstra is that on each iteration, we include the u with the smallest $r(u)$, and among the nodes with the same $r(u)$, we include the one with the smallest $dist(u)$.

Proof of correctness: We will argue when we add u to A, $dist(u)$ is the shortest path distance from s to u. Consider any $s \to u$ path we haven't considered (i.e. a path visits some node not in A before ending at u, let v be the first such node on the path). Since the algorithm adds u instead of v in this iteration, we must have either $r(v) > r(u)$ or $r(v) = r(u), dist(v) \geq dist(u)$. In the first case, there is no path from v to u (contradiction!), and in the second case, the part of the path from v to u must be all roads (thus positive), and $dist(v) \geq dist(u)$ guarantees the initial part of the path from s to v is already no shorter than the paths we have considered for $dist(u)$, thus the entire path won't be shorter. Either way, it is safe to say we have already computed the shortest path for u. Once we established this property, the correctness of the algorithm follows from the proof of Dijkstra.

Remark: Note that although Dijkstra's algorithm can not be used generally for finding shortest paths on a graph with negative edges, there are some special graph structure with negative edges (like this problem or problem 1 of this homework) that Dijkstra's algorithm still works.

Running time: it is the same as Dijkstra's running time, since the preprocessing (i.e. finding SCCs, lin-

earization) is linear time, and the rest is the same complexity as Dijkstra.

Rubric:

Problem 1, 20 pts

(a) 8 points: 2 points for each correctly ordered vertex (except s as it's specified as the starting vertex).

(b) 12 points
4 points: correctly identify that the goal is $distance(s,c) < distance(s,d) < distance(s,b)$
2 points: correctly calculate $distance(s,c)$ and $distance(s,b)$
4 points: correctly recognize the two possible cases for $distance(s,d)$
2 points: perform correct calculations for the inequalities with respect to these two cases and obtain the correct final result

Problem 2, 20 pts

- 5 pts for saying Dijkstra's Algorithm works
- 15 pts for a correct proof (contradiction or otherwise showing that the normal correctness of Dijkstra's Algorithm is unchanged.

Problem 3, 20 pts

- 5 pts Modified Dijkstra's Algorithm
- 15 pts
  if dist(v) = dist(u) + $l(u,v)$ **then**
  usp[v] = `false`
  end if

Problem 4, 20 pts

(a) 12 pts:
- 6 pts for reducing the problem to a shortest path problem.
- 6 pts for solving the problem with the Bellman-Ford algorithm, and showing why it works

(b) 8 pts:
- 4 pts for relating the presence of anomaly to a negative cycle in the graph
- 4 pts for showing how we can find negative cycle efficiently ($O(|V||E|)$, such as with the Bellman-Ford Algorithm)

Problem 5, 20 pts

(a) 6 pts for showing all internal h values cancel, leaving only $h(v_0)$ and $h(v_k)$, which are constant for all possible paths.

(b) 4 pts for showing all h values cancel in a cycle, preserving the original weight.

(c) 6 pts for using the triangle inequality to show the reweight formula cannot be negative.

(d) 4 pts for explaining the given algorithm works better for small $|E|$

Problem 6, 10 pts

(a) 5 pts for mentioning the SCCs of the graph and saying that all road edges lie entirely in SCCs, and plane edges only connect two different SCCs (plane edges do not lie in a SCC)

(b) 4 pts for correctly modifying the Dijkstra algorithm, and designing the efficient algorithm.

(c) 1 pts for showing the correctness of efficient algorithm.

CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 8

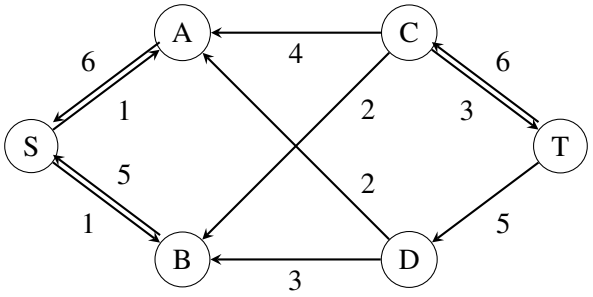
1. (20 pts.) **Network Flows.**

(a) The maximum flow is 11, with the following flows:

- 3 units on $S \to B \to D \to T$;
- 2 units on $S \to B \to C \to T$;
- 2 units on $S \to A \to D \to T$;
- 4 units on $S \to A \to C \to T$;

A minimum cut is between $\{S,A,B\}$ and $\{C,D,T\}$, with a value of 11. (The cut edges are $A \to C$, $A \to D$, $B \to D$, $B \to C$.)
Another minimum cut is between $\{S,A,B,D\}$ and $\{C,T\}$, with a value of 11. (The cut edges are $A \to C$, $B \to C$ and $D \to T$)

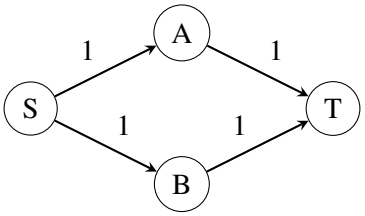
(b) The residual graph G_f is as follows:



$\{S,A,B\}$ are reachable from S, and from $\{C,T\}$ we can reach T.

(c) $A \to C$ and $B \to C$ are all crucial edges in the above graph because they are in every minimum cut of the graph. Note that increasing (A,D) or (B,D) does not change the value of the max flow, because doing so makes $[SABD,CT]$ the new min cut with value 11, so the max flow value does not change.

(d) The following graph does not have unique crucial edges:



(e) Run the usual network flow algorithm to get the residual graph G_f. In G_f, compute $\mathscr{S}$ as the vertices reachable from S, and $\mathscr{T}$ as the vertices which can reach T (by reversing edges in the residual graph), in linear time using any graph searching algorithm (BFS/DFS). An edge $u \to v$ is a crucial edge iff u is in $\mathscr{S}$ and v is in $\mathscr{T}$, and all of them can be identified in linear time.

2. (20 pts.) **Decreased Capacity.** Note that the maximum flow in the new network will be either f or $f-1$ (because changing one edge can change the capacity of the min-cut by at most 1). Now consider the residual graph of G, taking the capacity of edge (u,v) as c_{uv}. Since there is a flow of at least 1 unit going from v to t, the residual graph must have a path from t to v (each edge along which there is a flow creates a backward

edge in the residual graph). Similarly, there must be a path in the residual graph from u to s, since at least 1 unit of flow reaches u from s.

Find the paths by doing a DFS from u and t, and send back 1 unit of flow through this path. This changes the flow through edge (u,v) to $c_{uv}-1$. Notice that this is a valid flow even if we replace the capacity of edge (u,v) by $c_{uv}-1$. The flow through the new graph is now $f-1$, with the edge (u,v) having capacity $c_{uv}-1$ and a flow of $c_{uv}-1$ through it. This is just an intermediate stage of the Ford-Fulkerson algorithm. If it is possible to increase the flow, then there must be an $s-t$ path in the residual graph. This can be checked by a DFS (or BFS). Since the algorithm just involves calling DFS three times, the running time is $O(|V|+|E|)$

3. (20 pts.) **Edge-Disjoint Paths.**

(a) Simply set $C(e)$ to 1 for all edges in E. The correctness of this decision is shown in parts (b) and (c).

(b) Every edge-disjoint path from s to t can carry exactly 1 unit of flow given the choice of $C(e)=1$ from part (a). Additionally, because all of the k paths are edge-disjoint, there will be no conflicts if each path is assigned its maximum amount of flow. So, we can guarantee 1 unit from each of the k paths, for a total flow value of at least k.

(c) Assume Ford-Fulkerson has found a flow of value k. We will show that there must be at least k disjoint paths in G by using induction on the flow value k.
Base Case: Let $k=0$; the discovered flow function pushes no flow. Trivially, there are at least 0 edge-disjoint paths from s to t. Inductive Hypothesis: Assume that a flow of value $k-1$ implies that there must be at least $k-1$ edge-disjoint paths for every positive integer k. Inductive Step: Let k be any positive integer. Because k is positive, some flow must be pushed through some s-t path p. By the construction of our capacity function and the assumption that all flow values are integers, we know that the flow along p must be exactly 1. Note that we can ignore the possibility that there are cycles in this s-t path because Ford-Fulkerson does not produce cycles in its flow assignments. Knowing all of this, we can count p as one of our edge-disjoint paths. To find the other $k-1$ paths, we create a new flow f' by setting the flow along every edge of p to 0. This will remove the 1 unit of flow that was being pushed through p, meaning f' has a total flow value of $k-1$. At this point, the inductive hypothesis assures us that because f' is a flow with value $k-1$, there must be at least $k-1$ edge-disjoint paths in G. Adding p to that count gives us our final result that there are at least k edge-disjoint paths in G.

(d) The running time of Ford-Fulkerson discussed in class is $O(C(|V|+|E|))$, where C is the sum of the edge capacities out of s. In general, s can never have more than $O(|V|)$ outgoing edges, one to every other node, and here, every edge has capacity 1. Therefore, $C=O(|V|)$ and the total running time is $O(|V|^2+|V||E|)$.

4. (20 pts.) **Ford-Fulkerson with Irrational Numbers.**

(a) You can pass a units of flow from the path $s-V_1-t$, and a units from $s-V_4-t$. Also, we can pass 1 unit of flow from the path $s-V_2-V_3-t$, so the total amount of flow you can pass is $2a+1$

(b) The step of the Ford-Fulkerson algorithm are shown in the following table.
The final row of this table shows that the capacities of the residual edges are of the requested form after augmenting through path p_3.

(c) After step 1 as well as after step 5, the residual capacities of edges V_2-V_1, V_4-V_3 and V_2-V_3 are in the form r^n, r^{n+1} and 0, respectively, for some $n\in\mathbb{N}$. This means that we can use augmenting paths p_1, p_2, p_1 and p_3 infinitely many times and residual capacities of these edges will always be in the same form. The total flow in the network after step 5 is $1+2(r^1+r^2)$. If we continue to use augmenting paths as above, the total flow converges to $1+2\sum_{i=1}^{\infty}r^i=1+\sum_{i=1}^{\infty}2r^i=1+\sum_{i=1}^{\infty}(2r)r^{i-1}$.

Step	Augmenting path	Sent flow	Edge $V_2 - V_1$ in Residual graph	Edge $V_4 - V_3$ in Residual graph	Edge $V_2 - V_3$ in Residual graph
0			$r^0 = 1$	r	1
1	p_0	1	r^0	r^1	0
2	p_1	r^1	r^2	0	r^1
3	p_2	r^1	r^2	r^1	0
4	p_1	r^2	0	r^3	r^2
5	p_3	r^2	r^2	r^3	0

Table 1: Ford-Fulkerson iterations

This is a geometric series with $r < 1$, so we can use the convergence formula to say that it is equivalent to $1 + \frac{2r}{1-r}$. To arrive at the final simplified form, we can use several facts stemming from $r = \phi - 1$, or one less than the golden ratio. It is known that $\phi - 1 = \frac{1}{\phi}$ and $\phi^2 = \phi + 1$. This is helpful because it can then be shown that $\frac{1}{1-r} = r + 2$, which gives us $1 + 2r(r+2) = 1 + 4r + 2r^2$. It can also be shown that $r^2 = 1 - r$, which gives us our final value of $3 + 2r$.

5. (20 pts.) **Matching Rooks.**

(a) Let $r_1, \ldots, r_m$, and $c_1, \ldots, c_n$ to be the vertices in G, representing the m rows, and n columns of chessboard. Also, let s and t be two other nodes to act as the source and sink in the flow network. First, we define the edges of G. Connect vertex s to all vertices in $r_1, \ldots, r_m$ (this gives us m edges), and their capacities to one. Then, for all $1 \leq i \leq m$, and $1 \leq j \leq n$, create an edge between r_i and c_j, and set the capacity to zero if the square at row i and column j^{th} of chessboard is unusable, otherwise set the capacity to one. Finally, connect all $c_1, \ldots, c_n$ to t, and set their capacities to one. Now we can solve this network using any max flow algorithm. Note that a path $s \to r_i \to c_j \to t$ carrying 1 unit of flow corresponds to putting a rook in the square at row i and column j of the chessboard. In other words, we can solve the problem by examining the max flow and placing a rook on square $(i,\ j)$ if the flow between r_i and c_j is 1. Also observe that the incoming flow for each of the row vertices $(r_1, \ldots, r_m)$ is at most one, and the outgoing flow for each of the column vertices $(c_1, \ldots, c_n)$ is at most one, this means that given an integer flow, for any i, r_i is connected to at most one of the column vertices (by connection here we mean an edge with flow of 1), and for any j, c_j is connected to at most one of the row vertices. This property ensures that there is at most one rook in each row and in each column, hence the max flow will give us a valid rook placement. As a result, to solve the matching rooks problem, it suffices to find the max flow of the given graph.

(b) Using the flow network we discussed in part (a), we run the Ford-Fulkerson algorithm until we reach a maximum flow. Note that the value of the maximum flow is the maximum number of rooks we can place on the chessboard. Since we can place at most one rook in any row or column, this number is at most $\min(m,n)$. Finding each augmenting path takes $O(|V| + |E|) = O(mn)$ since there are at most mn edges and $m + n + 2$ vertices in the graph. So, the total running time of Ford-Fulkerson would be $O(f_{\max}(|V| + |E|)) = O(\min(m,n)(n + m + 2) + \min(m,n)mn) = O(\min(m,n)mn)$.

Rubric:

Problem 1, 20 pts

(a) total 6 pts
- 1pt for every path
- 2pts for the minimum cut

(b) total 4 pts
- 2pt for residual graph
- 1pt for reachable vertices from S
- 1pt for vertices that can reach T

(c) 3pt for $A \to B$ and 1pts for $B \to C$

(d) 3pts for any correct graph that has no crucial

(e) 4pts for correct algorithm

Problem 2, 20 pts

- 10 pts for backward edges in the residual graph and decreasing the proper edge's flow values by 1.
- 10 pts for running 1 step of Ford Fulkerson (looking for an *s*-*t* path and increasing the flow by 1 if found).

Problem 3, 20 pts

1. 3 pts for $C(e) = 1$ for all edges.
2. 5 pts for explaining that each disjoint path can carry 1 unit of flow.
3. 7 pts for an inductive proof (base case, hypothesis, inductive step) or any other valid proof.
4. 5 pts. 2 for general FF running time, 3 for simplifying with $C = O(|V|)$

Problem 4, 20 pts

(a) 6 pts for a correct maximum flow.

(b) 7 pts for showing the pattern in some way

(c) 7 pts for giving the FF max flow as a function of r and simplifying (be lenient many students will struggle with the algebra and may argue based on computing some values with a calculator).

Problem 5, 20 pts

(a) 14 pts: 7 pts for defining edges for flow network. 7 pts for explaining how the flow network can be used to solve the matching rooks problem.

(b) 6 pts: 2 pts for using augmentation path or Ford–Fulkerson method to solve the max flow network. 4 pts for showing the running time. For running time, mn^2, or m^2n are also acceptable.

CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 9

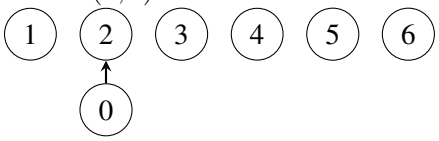
1. (20 pts.) **Kruskal's Algorithm.** The order of edges that are added to the MST is as below with the current set(s):

1. (d,f); current sets: $\{a\},\{b\},\{c\},\{d,f\},\{e\}$
2. (a,b); current sets: $\{a,b\},\{c\},\{d,f\},\{e\}$
3. (b,e); current sets: $\{a,b,e\},\{c\},\{d,f\}$
4. (d,e); current sets: $\{a,b,e,d,f\},\{c\}$
5. (e,c); current set: $\{a,b,e,d,f,c\}$

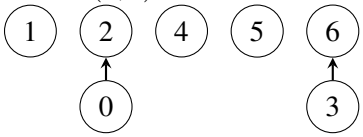
2. (20 pts.) **Disjoint Set Operations.**

(a) Initial:

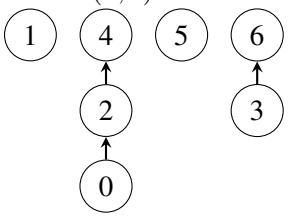
`union`$(0,2)$:



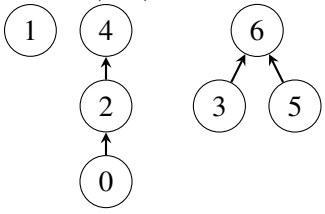
`union`$(3,6)$:



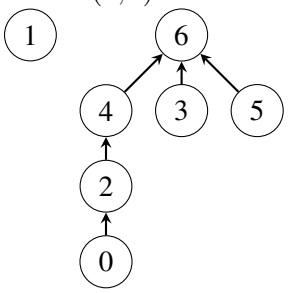
`union`$(2,4)$:



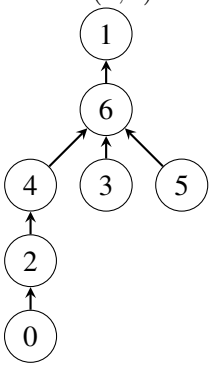
`union`$(5,3)$:



`union`(0,6):

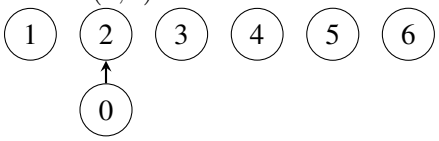


`union`(5,1):

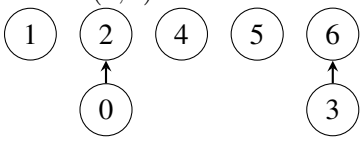


(b) Initial:

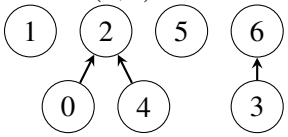
`union`(0,2):



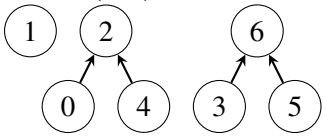
`union`(3,6):



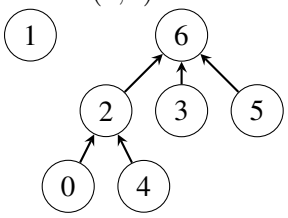
`union`(2,4):



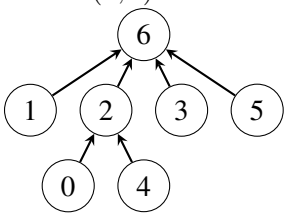
`union`(5,3):



`union`$(0,6)$:



`union`$(5,1)$:



(c) `root` must visit each node between the desired node (here 0) and the root, inclusive. Looking at part (a)'s final tree, this requires visiting 5 of the 7 nodes, (0, 2, 4, 6, and 1). Part (b)'s final tree requires only 3 of the 7 nodes to be visited. (0, 2, and 6).

(d) The worst-case series of unions for part (a) turns the final tree into a single branch, for example, `union`$(0,1)$, `union`$(1,2)$, `union`$(2,3)$, etc. Because every node is on a single path from the root, running `root` on the bottom-most node requires visiting all 7 nodes. However, the worst-case series of unions for part (b) is less clear. Because any merged tree is placed as a direct child of the root in the tree it's merging into, the height can only be increased when merging two trees of equal heights. So, creating a height 3 tree requires at least merging two height 2 trees, or 4 nodes total. Therefore, we do not have enough nodes to construct a height 4 tree, which would require at least 2 height 3 trees, or 8 nodes total. So the worst possible number of nodes on the path from a leaf to the root is 3 for part (b).

3. (20 pts.) **Fractional Knapsack.**

(a) First, we sort the items in descending order of the ratio $\frac{v_i}{w_i}$. Then, we pick items from the beginning of the sorted list until we're either out of items (in which case we're done) or our knapsack does not have enough remaining weight to add the next item. In the latter case, we take as much of the next item as we can until our total weight is exactly W. The running time of this algorithm is dominated by the time it takes to sort the items, which is $O(n \log n)$. The correctness of this algorithm is discussed in part (b).

(b) This greedy algorithm relies on the assumption that picking the item with the best value-to-weight ratio will always be an optimal choice. In the 0-1 Knapsack problem, it is easy to create an example where this assumption fails. For example, $W = 10$, $(v_1, w_1) = (10, 5)$, and $(v_2, w_2) = (11, 10)$. Item 2 has a worse ratio, but picking item 1 uses only half the available weight, which allows item 2 to achieve a higher total value. Generalizing this idea, we can say that greedily selecting items based only on their v_i and w_i does not always arrive at the optimal solution, as the amount of remaining weight in the knapsack and the other possible selections must be taken into account at each step. In contrast, the Fractional Knapsack allows us to always maximize the weight we select to be equal to W. This removes the possibility that a greedy choice might block better future choices. Any optimal selection must use the entire weight W since all the values are positive. Therefore, the greedy solution that picks the highest value-to-weight ratio, multiplied by W, results in the highest possible value for a weight W knapsack.

4. (20 pts.) **Largest k-Degree Subgraph.** If a node has degree less than k, then neither it nor its edges can be included in G'. Thus, if v has degree less than k, then $G' \subseteq G - v$. This implies the correctness of the following "greedy" algorithm: while G has a node v of degree less than k, set $G := G - v$ and repeat.

To implement this, we first calculate the degree of every node in the graph by finding the number of nodes in each node's adjacency list. Then, we construct a list of nodes with degree less than k by iterating over every node and checking the precomputed degree value. While the list of nodes with degree less than k is not empty, take an element from the list, remove it from the graph, subtract one from the degree of all of its neighbors, and if any of them now have degree less than k, add them to the list. When the list is empty, G' is comprised of the nodes remaining in the graph and the edges between them in E.

This algorithm takes $O(|V| + |E|)$ steps, since each vertex is examined at most once, and there may be a number of subtractions equal to the number of edges.

5. (20 pts.) **MST Edge Reweighting.**

(a) Since the change only increases the cost of some other spanning trees (those including e) and the cost of T is unchanged, it is still an MST.

(b) We include e in the tree, thus creating a cycle. We then remove the heaviest edge e' in the cycle, which can be found in linear time, to get a new tree T'. To prove this, we argue that T' contains a least-weight edge across every cut of G and is hence an MST. Note that since the only changed edge is e, $T \cup \{e\}$ already includes a least-weight edge across every cut. We only removed e' from this. However, any cut crossed by e', must also be crossed by at least one more edge of the cycle, which must have weight less than or equal to e'. Since this edge is still present in T', it contains a least-weight edge across every cut. Thus, T' is an MST.

(c) The tree is still an MST if the weight of an edge in the tree is reduced. Hence, no changes are required.

(d) We remove e from the tree to obtain two components, and hence a cut. We then include the lightest edge across the cut to get a new tree T'. We can now "build up" T' using the cut property to show that it is an MST. Let $X \subseteq T'$ be a set of edges that is part of some MST, and let $e_1 \in T' \setminus X$. Then, $T' \setminus \{e_1\}$ gives a cut which is not crossed by any edge of X and across which e_1 is the lightest edge. Hence, $X \cup \{e_1\}$ is also a part of some MST. Continuing this, we can grow X to $X = T'$, which must be then an MST.

Rubric:

Problem 1, 20 pts

- 2 pts for each correct edge stated in the correct order
- 2 pts per step for the vertex sets (order of elements in each set does not matter)

Problem 2, 20 pts

(a) 6pts, 1 for each union forest.

(b) 6pts, 1 for each union forest.

(c) 4pts for the correct answer, (a) requires 5, (b) requires 3.

(d) 4pts for the correct answer, (a) requires at most 7, (b) requires at most 3.

Problem 3, 20 pts

(a) 12 pts for the correct ratio-based algorithm.

(b) 8 pts for explaining 0-1 can waste space, but Fractional always fills the space, so best ratio is optimal.

Problem 4, 20 pts

- 12 pts for designing correct algorithm
- 4 pts for proof of correctness (just one or two sentences would be enough for this problem)
- 4 pts for showing why the algorithm runs in $O(|V|+|E|)$. Note that the algorithm should be correct in order to get the point of this part.

Problem 5, 20 pts

(a) this part is worth 4 pts, only saying that ”it is still an MST” is enough.

(b) this part is worth 6 pts, designing correct algorithm (or equivalently updating MST tree correctly) is worth 4 pts, and proof of correctness is worth 2 pts.

(c) this part is worth 4 pts, only saying that ”no update is needed” would be enough.

(d) this part is worth 6 pts, designing correct algorithm (or equivalently updating MST tree correctly) is worth 4 pts, and proof of correctness is worth 2 pts.

CMPSC 465 Fall 2023 — Data Structures & Algorithms — Antonio Blanca and Yana Safonova — HW 10

1. (20 pts.) **Set Cover.**

1. The first selected subset is $\{t,h,r,e,a,d\}$. As all the letters are uncovered elements, we simply pick the set with the most letters (six).
2. The next selected subset is $\{l,o,s,t\}$ because it has three uncovered elements, which is the most compared to other words.
3. The third subset we pick is $\{a,f,r,i,d\}$. While this subset, $\{d,r,a,i,n\}$, and $\{s,h,u,n\}$ all have two uncovered elements i.e. the most currently, $\{a,f,r,i,d\}$ appears first and should be selected based on the tie-breaking rule.
4. The last selected subset is $\{s,h,u,n\}$ since it's the only subset that still has two uncovered elements. After this, all letters/elements are covered.

2. (20 pts.) **Huffman Encoding.**

(a) It's possible to obtain this sequence. Any f_a, f_b, f_c such that $f_a > f_b \geq f_c$ can result in this encoding.

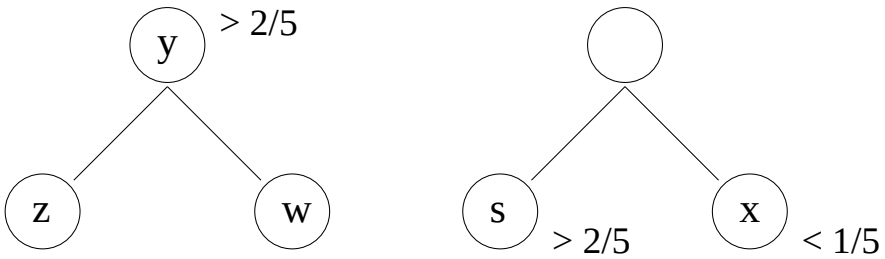
(b) This encoding is not possible since the encoding for a (1), is a prefix of the encoding for c (10).

(c) This encoding also cannot possibly be obtained as it's suboptimal. The encoding for one of the letters in the alphabet could have length 1. For example, the encoding for a could be 0 so that we don't waste space.

3. (20 pts.) **Huffman Properties.**

a Let s be the symbol with the highest frequency (probability) $p(s) > 2/5$ and suppose that it merges with some other symbol during the process of constructing the tree and hence does not correspond to a codeword of length 1. To be merged with some node, the node s and some other node x must be the two with minimum frequencies. This means there was at least one other node y (formed by merging of other nodes), with $p(y) > p(s)$ and $p(y) > p(x)$. Thus, $p(y) > 2/5$ and hence $p(x) < 1/5$.

Now, y must have been formed by merging some two nodes z and w with at least one of them having probability greater than $1/5$ (as they add up to more than 2/5). But this is a contradiction - $p(z)$ and $p(w)$ could not have been the minimum since $p(x) < 1/5$.



b Suppose this is not the case. Let x be a node corresponding to a single character with $p(x) < 1/3$ such that the encoding of x is of length 1. Then x must not merge with any other node till the end. Consider the stage when there are only three leaves - x, y and z left in the tree. At the last stage y, z must merge to form another node so that x still corresponds to a codeword of length 1. But, $p(x) + p(y) + p(z) = 1$

and $p(x) < 1/3$ implies $p(y) + p(z) > 2/3$. Hence, at least one of $p(y)$ or $p(z)$, say $p(z)$, must be greater than $1/3$. But then these two cannot merge since $p(x)$ and $p(y)$ would be the minimum. This leads to a contradiction.

4. (20 pts.) **Feedback Edge Set.** In any connected graph, removing the minimum-weight feedback edge set will leave a spanning tree. This must be the case because a spanning tree has the most edges of any acyclic graph, and a minimum-weight feedback edge set will not remove extra edges after the graph has already become acyclic. So, if G is connected, we can compute the minimum-weight feedback edge set by choosing every edge not in the maximum-weight spanning tree. This requires running Kruskal's algorithm (or Prim's) with every edge weight multiplied by -1 (or equivalently simply modifying the algorithm to choose the heaviest available edge), which has running time $O(|E|\log|V|)$.

If G is not connected, we can repeatedly apply the above approach to construct our final feedback edge set. First, run DFS on G to separate G's edges into connected components in $O(|V|+|E|)$ time. Then, apply the above approach to each connected component. Because E edges are still being considered by Kruskal's algorithm in total, the running time of the entire algorithm is $O(|V|+|E|\log|V|)$. The algorithm is correct because it leaves the heaviest edges that preserve G's connected components while removing all of the lighter edges that make up G's cycles.

Note that Kruskal's Algorithm does not necessarily fail on a disconnected graph. It would create several maximum-weight trees, each of which span a connected component. Therefore, mentioning this fact and running Kruskal's Algorithm a single time is another acceptable solution that runs in $O(|E|\log|V|)$.

5. (20 pts.) **Huffman Efficiency.**

(a) $m\log_2(n) = mk$ bits.

(b) The efficiency is smallest when all characters appear with equal (or near-equal) frequency. In this case, the binary tree that represents the encoding is a complete tree and each encoding takes $\log n = k$ bits. Therefore, encoding the entire file takes mk bits and $E(F) = 1$.

(c) Let F be $x_0, x_1, \ldots, x_{n-2}$ followed by $m-(n-1)$ instances of x_{n-1}. x_{n-1} will be encoded as a single bit, with all of the others being approximately $\log_2(n)+1$ bits (because n is a power of 2, one of the infrequent symbols will be encoded using $\log_2(n)$ bits, but this minor difference will be lost in the big-O notation). This file has efficiency:

$$\frac{m\log_2(n)}{(m-(n-1))\cdot 1+(n-1)\cdot(\log_2(n)+1)} = \frac{m\log_2(n)}{m-(n-1)+(n-1)\log_2(n)+(n-1)}$$

Assuming m is very large, we have the following big-O notation:

$$= \frac{m\log_2(n)}{O(m)+O(n\log n)} = \frac{m\log_2(n)}{O(m)} = O(\log(n))$$

So the efficiency is $O(\log(n)) = O(k)$

Rubric:

Problem 1, 20 pts

- 3 points: for each correct selected subset.
- 2 points: for correctly identifying the number of uncovered elements in each subset.

Problem 2, 20 pts

(a) 3 points: correct conclusion
3 points: correct set of frequencies

(b) 3 points: correct conclusion
4 points: correct explanation i.e. correctly point out the issue

(c) 3 points: correct conclusion
4 points: correct explanation i.e. correctly point out the issue

Problem 3, 20 pts

(a) 10 pts for right proof

(b) 10 pts for right proof

Problem 4, 20 pts

- 6 points for $E - E'$ is a maximum spanning tree in a connected graph.
- 4 points for how to find the maximum spanning tree.
- 6 points for generalizing to an unconnected graph (Arguing how Kruskal functions properly on the unconnected graph is also valid).
- 4 points for running time analysis.

Problem 5, 20 pts

(a) 6 pts for correct answer.

(b) 6 pts: 3 for $E(f) = 1$, 3 for all frequencies equal.

(c) 8 pts: 4 for correct frequencies, 4 for big-O analysis.

CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 11

1. (20 pts.) **Evacuation Dynamic Program.**

1. The subproblems are the number of paths to every intermediate node. To implement the solution, you would make an array A that tracks the number of paths to every node between the start (index 0) and end (final index) in topological order.
2. The base case is the number of paths to the starting node, which is 1. $A[0] = 1$.
3. $A[j] = \sum_{(i,j)\in E} A[i] \qquad \forall j \geq 1$

2. (25 pts.) **Weighted Set Cover.** As in the unweighted case, we will use a greedy algorithm:

```
while(some element of (B) is not covered)
{
    Pick the set (S_i) with the largest ratio
    ((new elements covered by (S_i)) /w_i).
}
```

Now we will prove that if there is a solution of cost k, then the above greedy algorithm will find a solution with cost at most $k \log_e n$. Our proof is similar to the proof in the unweighted case in Section 5.4 of the textbook.

After t iterations of the algorithm, let n_t be the number of elements still not covered, and let k_t be the total weight of the sets the algorithm chose, so $n_0 = n$ and $k_0 = 0$. Since the remaining n_t elements are covered by a collection of sets with cost k, there must be some set S_i such that S_i covers at least $w_i n_t/k$ new elements. (This is easiest to see by contradiction: if every set S_i covers less than $w_i n_t/k$ elements, then any collection with total weight k will cover less than kn_t/k elements.) Because some set with such a desirable ratio exists, our greedy algorithm can pick that set to ensure that it is always picking a set with ratio at least $w_i n_t/k$. Therefore the greedy strategy will ensure that

$$n_{t+1} \leq n_t - \frac{n_t(k_{t+1} - k_t)}{k} = n_t(1 - (k_{t+1} - k_t)/k).$$

Now, we apply the fact that for any x, $1 - x \leq e^{-x}$, with equality iff $x = 0$:

$$n_{t+1} < n_t e^{-(k_{t+1} - k_t)/k}.$$

By induction, we find that for $t > 0$, $n_t < n_0 e^{-k_t/k}$. If we choose the smallest t such that $k_t \geq k \log_e n$, then, n_t is strictly less than 1, which means no elements remain to be covered after t steps. Note that we will never add a set of weight more than k, due to the relationship discussed in paragraph 2. There exists some set that covers at least $w_i n_t/k$ elements, which implies that set's ratio is at least n_t/k. Therefore, the only time we can choose a set with weight k is if it covers all remaining elements, and it is not possible to choose a set with weight more than k. It follows that $k_t < k \log_e n + k = O(k \log n)$.

3. (25 pts.) **Exponential Greedy Solutions.** To create exponentially many possible solutions, we could construct an instance that encounters a tie between a constant number of elements at every step and runs for $O(n)$ steps. However, the easier approach is to construct an instance that encounters a single tie between an

exponential number of sets. Let the set of items $U = \{1, 2, \ldots, n\}$. We first construct a set $S_1 = \{1, 2, \ldots, n-1\}$. We want the greedy algorithm to pick S_1 first. Then, we want any other set to be able to cover item n in the second step of the algorithm, creating a tie between all remaining sets. Now we construct an exponential number of other subsets S_i that all cover n, while making sure none of them become a better first option than S_1. To keep things simple, let's accomplish this by saying none of the S_i include items 1 or 2, but all of them include item n. This leaves $n-3$ items, and we will construct one set for every permutation of including and excluding those items, for $2^{n-3} = O(2^n)$ sets total. At the second step of the greedy algorithm, there will be a tie between all $O(2^n)$ sets, creating $O(2^n)$ possible greedy solutions.

4. (25 pts.) **Longest Common Substring.** *Algorithm:* For $0 \leq i \leq n$ and $0 \leq j \leq m$, define $L(i, j)$ to be the length of the longest common postfix of $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$. When $i = 0$, it indicates $x_1 x_2 \ldots x_i$ is an empty string. Similarly, when $j = 0$, it indicates $y_1 y_2 \ldots y_j$ is an empty string. The recursion is:

$$L(i, j) = \begin{cases} L(i-1, j-1) + 1 & \text{if } x_i = y_j \\ 0 & \text{otherwise} \end{cases}$$

The initialization is $L(0, 0) = 0$. Then, for all $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$L(i, 0) = 0$$
$$L(0, j) = 0$$

The length of the longest common substring of x and y is the maximum of $L(i, j)$ over all $1 \leq i \leq n$ and $1 \leq j \leq m$. We can find the maximum by either tracking the maximum when solving all $L(i, j)$ or going through all those values when they are calculated.

Correctness and Running Time: The initialization is clearly correct as there is no postfix for an empty string. The heuristic to solve this problem is that the longest common substring of x and y must be the longest common postfix among all possible substrings of x and all possible substrings of y. However, we don't need to go through all substrings of x and y since we only care about the postfixes i.e. the ending parts. Thus, going through all possible combinations of $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$ is enough.
This implies we have $O(mn)$ subproblems regarding $L(i, j)$. Since each takes constant time to evaluate if we have the previous results (which will be the case as we use dynamic programming), the time to solve all subproblems is $O(mn)$. Either method of finding the maximum keeps the overall running time at $O(mn)$.

Alternative Solution:
Algorithm: For $1 \leq i \leq (n+1)$ and $1 \leq j \leq (m+1)$, define $L(i, j)$ to be the length of the longest common prefix of $x_i x_{i+1} \ldots x_n$ and $y_j y_{j+1} \ldots y_m$. When $i = n+1$, it indicates $x_i x_{i+1} \ldots x_n$ is an empty string. Similarly, when $j = m+1$, it indicates $y_j y_{j+1} \ldots y_m$ is an empty string. The recursion is:

$$L(i, j) = \begin{cases} L(i+1, j+1) + 1 & \text{if } x_i = y_j \\ 0 & \text{otherwise} \end{cases}$$

The initialization is $L(n+1, m+1) = 0$. Then, for all $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$L(i, m+1) = 0$$
$$L(n+1, j) = 0$$

The length of the longest common substring of x and y is the maximum of $L(i, j)$ over all $1 \leq i \leq n$ (starting from n going back to 1) and $1 \leq j \leq m$ (starting from m going back to 1). We can find the maximum by either tracking the maximum when solving all $L(i, j)$ or going through all those values when they are calculated.

Correctness and Running Time: The initialization is clearly correct as there is no prefix for an empty string. The heuristic to solve this problem is that the longest common substring of x and y must be the longest common prefix among all possible substrings of x and all possible substrings of y. However, we don't need to go through all substrings of x and y since we only care about the prefixes i.e. the beginning parts. Thus, going through all possible combinations of $x_i x_{i+1} \dots x_n$ and $y_j y_{j+1} \dots y_m$ is enough.
This implies we have $O(mn)$ subproblems regarding $L(i, j)$. Since each takes constant time to evaluate if we have the previous results (which will be the case as we use dynamic programming), the time to solve all subproblems is $O(mn)$. Either method of finding the maximum keeps the overall running time at $O(mn)$.

Rubric:

Problem 1, 25 pts

Directly from solutions to homework 6.

(a) 8 pts

(b) 8 pts

(c) 9 pts

Problem 2, 25 pts

- 10 pts for giving the algorithm including:

```
while(some element of (B) is not covered
{
    Pick the set (S_i) with the largest ratio
    ((new elements covered by (S_i)) /w_i).
}
```

- 15 pts for proving the algorithm

Problem 3, 25 pts

- 5 pts for identifying at least one case where ties lead to an exponential number of solutions
- 20 pts for the set cover instance (might stack many ties instead of one huge tie).

Problem 4, 25 pts

- 5 points: subproblems
- 5 points: recurrence relation
- 2 points: initialization
- 2 points: retrieving final answer from computed DP
- 5 points: proof of correctness
- 5 points: running time analysis

CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 12

1. (20 pts.) **Maximum Sum Contiguous Subsequence.**

Subproblems: Define an array of subproblems $D(i)$ for $0 \leq i \leq n$. $D(i)$ will be the largest sum of a (possibly empty) contiguous subsequence ending exactly at position i.

Algorithm and Recursion: The algorithm will initialize $D(0) = 0$ and update the $D(i)$'s in ascending order according to the rule:

$$D(i) = \max\{0, D(i-1) + a_i\}$$

The largest sum is then given by the maximum element $D(i)^*$ in the array D. The contiguous subsequence of maximum sum will terminate at i^*. Its beginning will be at the first index $j \leq i^*$ such that $D(j-1) = 0$, as this implies that extending the sequence before j will only decrease its sum.

Correctness: The contiguous subsequence of largest sum ending at i will either be empty or contain a_i. In the first case, the value of the sum will be 0. In the second case, it will be the sum of a_i and the best sum we can get ending at $i-1$, i.e. $D(i-1) + a_i$. Because we are looking for the largest sum, $D(i)$ will be the maximum of these two possibilities.

Running Time: The running time for this algorithm is $O(n)$, as we have n subproblems and the solution of each can be computed in constant time. Moreover, the identification of the optimal subsequence only requires a single $O(n)$ time pass through the array D.

2. (20 pts.) **Maximum Average Value Path.** One interesting observation is we need N-1 down moves and N-1 right moves to reach the destination (bottom right). So any path from the top left corner to the bottom right corner requires 2N – 2 cells. In the average value ratio, the denominator is fixed and we need to maximize the numerator. Therefore we need to find the maximum sum path. If dp[i][j] represents the maximum sum till cell (i, j) from (0, 0) then at each cell (i, j), we update dp[i][j] as below:

(i) For j = 0, the only possible way to reach this cell is by moving down. Thus,
dp[i][0] = dp[i-1][0] + cost[i][0] for all i from 1 to n.

(ii) For i =0, the only possible way to reach this cell is by moving right. Thus,
dp[0][j] = dp[0][j-1] + cost[0][j] for all j from 1 to n.

(iii) For the inner cells,
dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + cost[i][j]

Once we get the maximum sum of all paths, we divide this sum by (2N – 2) to get the maximum average. This algorithm performs a constant time operation for each cell of the N by N matrix, so it runs in $O(N^2)$ time. This algorithm is correct because there are only two ways to enter any interior cell. Recursively, the maximum value possible to reach some cell must be the maximum value of either its top neighbor or its left neighbor, plus its own value.

3. (20 pts.) **Coin Toss.**

Subproblems: Define $L(i, j)$ to be the probability of obtaining exactly j heads amongst the first i coin tosses.

Algorithm and Recursion: By the definition of L and the independence of the tosses:

$$L(i,j) = p_i L(i-1,j-1) + (1-p_i)L(i-1,j) \quad j = 0,1,\ldots,i$$

We can then compute all $L(i,j)$ by first initializing $L(0,0) = 1$, $L(i,j) = 0$ for all $j < 0$, and $L(i,j) = 0$ for all $i = 0, j \geq 1$. Then we proceed incrementally (in the order $i = 1,2,\ldots,n$, with inner loop $j = 0,1,\ldots,i$). The final answer is given by $L(n,k)$.

Correctness and Running Time: The recursion is correct as we can get j heads in i coin tosses either by obtaining $j-1$ heads in the first $i-1$ coin tosses and throwing a head on the last coin, which takes place with probability $p_i L(i-1,j-1)$, or by having already j heads after $i-1$ tosses and throwing a tail last, which has probability $(1-p_i)L(i-1,j)$. Besides, these two events are disjoint, so the sum of their probabilities equals $L(i,j)$. Finally, computing each subproblem takes constant time, so the algorithm runs in $O(nk)$ time.

4. (20 pts.) **Longest Palindromic Subsequence.** *Subproblems:* Define variables $L(i,j)$ for all $1 \leq i \leq j \leq n$ so that, in the course of the algorithm, each $L(i,j)$ is assigned the length of the longest palindromic subsequence of string $a[i,\cdots,j]$.

Algorithm and Recursion: The recursion will then be:

$$L(i,j) = \max\left\{L(i+1,j), L(i,j-1), L(i+1,j-1) + \text{equal}(a_i,a_j)\right\}$$

where equal(a,b) is 2 if a and b are the same character and is 0 otherwise, The initialization is the following:

$$\begin{aligned} \forall i, 1 \leq i \leq n \ , &\qquad L(i,i) = 1 \\ \forall i, 1 \leq i \leq n-1 \ , &\quad L(i,i+1) = 2 \text{ if } a_i = a_{i+1}, \text{ else } 1 \end{aligned}$$

$L(1,n)$ is the length of the longest palindromic subsequence.

Correctness and Running Time: Consider the longest palindromic subsequence s of $a[i,\cdots,j]$ and focus on the elements a_i and a_j. There are then three possible cases:

- If both a_i and a_j are in s then they must be equal and $L(i,j) = L(i+1,j-1) + \text{equal}(a_i,a_j)$
- If a_i is not a part of s, then $L(i,j) = L(i+1,j)$.
- If a_j is not a part of s, then $L(i,j) = L(i,j-1)$.

Hence, the recursion handles all possible cases correctly. The running time of this algorithm is $O(n^2)$, as there are $O(n^2)$ subproblems and each takes $O(1)$ time to evaluate according to our recursion.

5. (20 pts.) **Semi-Global Sequence Alignment.**

(a) We want S_1 to be able to start anywhere, without penalty. This can be translated to $E(0,j) = 0$ for all $i = 0,\ldots n$, or the top row of the algorithm's table being initialized to zero instead of counting up from zero to n. This allows the algorithm to begin placing characters from S_1 (moving down the rows of the table) aligned with any character of S_2 (in any column) with no penalty. Because it is only a change to the initialization values of the edit distance algorithm, the running time and correctness of the recurrence relation are unaffected.

(b) Similar to (a), but we instead want the leftmost column to be initialized to zero. This allows us to start placing letters from S_2 in any row without penalty.

(c) We want S_1 to be able to end in the middle of S_2, instead of both strings needing to end together. The algorithm knows that S_1 has ended when the final row has been reached, which can happen in any column thanks to the new rule. So, instead of recovering our final answer from the bottom-rightmost cell, we look for the minimum value from any cell of the bottom row. Taking the minimum ensures that no penalties after reaching the end of S_1 are counted. This change only affects how the final answer is retrieved, changing that step from $O(1)$ to $O(m)$. However, this is dominated by the algorithm's $O(nm)$ running time. The correctness of the rest of the algorithm is unchanged.

(d) Similar to (c), but we instead iterate over all the cells in the rightmost column. This will give us the minimum edit distance counting only until S_2 has ended and takes $O(n)$ time. The rest of the answer is identical to (c).

Rubric:

Problem 1, 20 pts

- 4 pts: subproblem definition
- 4 pts: base case
- 8 pts: right recursion
- 2 pts: correctness proof
- 2 pts: running time analysis

Problem 2, 20 pts

- 4 pts: subproblem definition
- 4 pts: base case (left column, top row)
- 8 pts: right recursion
- 2 pts: correctness proof
- 2 pts: running time analysis

Problem 3, 20 pts

- 4 pts: subproblem definition
- 4 pts: base case
- 8 pts: right recursion
- 2 pts: correctness proof
- 2 pts: running time analysis

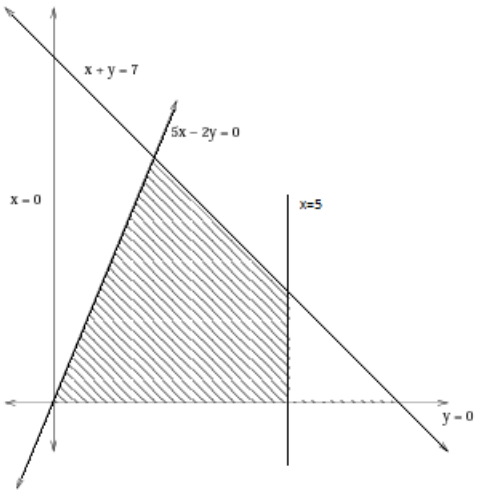
Problem 4, 20 pts

- 4 pts: subproblem definition
- 4 pts: correct base case (2 points for $L(i,i)$ case and $L(i,i+1)$ case, respectively)
- 8 pts: correctly identify all possible cases for the subproblem (3 points for each possible case of $L(i,j)$: $L(i+1,j)$, $L(i,j-1)$, and $L(i+1,j-1)+\text{equal}(a_i,a_j)$)
- 3 pts: explains the correctness of each of the 3 possible cases (1 point each)
- 1 pts: running time analysis

Problem 5, 20 pts All parts 5pts. 3 for the modification, 2 for a reasonable explanation.

1. (20 pts.) **Solving a Linear Program.**

(a) The feasible solution is the shaded region in the below graph:



(b) The new linear program uses a surplus variable a_1 for the first constraint and slack variables a_2 and a_3 for the second and third.

$$\begin{aligned} \text{maximize} \quad & 5x+3y \\ \text{subject to} \quad & 5x-2y-a_1=0 \\ & x+y+a_2=7 \\ & x+a_3=5 \\ & x \geq 0 \\ & y \geq 0 \\ & a_1 \geq 0 \\ & a_2 \geq 0 \\ & a_3 \geq 0 \end{aligned}$$

(c) The optimal solution is achieved in the upper right corner of the feasible region at the point $(5,2)$, and has a value of $5x+3y=31$. The initial tableau is shown below.

	x	y	a_1	a_2	a_3	b
s_1	5	-2	-1	0	0	0
s_2	1	1	0	1	0	7
s_3	1	0	0	0	1	5
z	-5	-3	0	0	0	0

Step 1: Because of the negative entry in column a_1, we first select the farthest left column with a positive entry in the row containing the -1 as the pivot column, which here is column x with a value

of 5. The smallest ratio of b column value to x column value is in row s_1 with a ratio of 0, so row s_1 is the pivot row. We now rescale that row so the 5 becomes 1 and perform elimination:

	x	y	a_1	a_2	a_3	b
s_1	1	-0.4	-0.2	0	0	0
s_2	0	1.4	0.2	1	0	7
s_3	0	0.4	0.2	0	1	5
z	0	-5	-1	0	0	0

Step 2: We now return to the normal pivot selection procedure. The column with the smallest value in row z is column y. The second pivot row is s_2 with the smallest ratio of 5. We now rescale that row so the 1.4 becomes 1 and perform elimination:

	x	y	a_1	a_2	a_3	b
s_1	1	0	-1/7	2/7	0	2
s_2	0	1	1/7	5/7	0	5
s_3	0	0	1/7	-2/7	1	3
z	0	0	-2/7	25/7	0	25

Step 3: The column with the smallest value in row z is column a_1. The third pivot row is s_3 because the other rows have already been used as pivots. We now rescale that row so the $1/7$ becomes 1 and perform elimination (here it's easier to eliminate first, then rescale):

	x	y	a_1	a_2	a_3	b
s_1	1	0	0	0	1	5
s_2	0	1	0	1	-1	2
s_3	0	0	1	-2	7	21
z	0	0	0	3	2	31

The maximum objective function value is the entry in row z, column b: 31. The assignment that produces that optimal value are the b values corresponding to the pivots (cells of value 1) in columns x and y: $x = 5$ and $y = 2$.

(d) The dual linear program is:

$$\begin{aligned} \text{minimize} \quad & 7b + 5c \\ \text{subject to} \quad & 5a + b + c \geq 5 \\ & -2a + b \geq 3 \\ & a \leq 0 \\ & b \geq 0 \\ & c \geq 0 \end{aligned}$$

This is found by multiplying the first three constraints (the constraints that are for more than just non-negativity) by a, b, and c, respectively. Note that $a \leq 0$ comes from the fact that the first constraint uses $\geq$ instead of $\leq$ in the primal. Note that the counts of variables and interesting constraints flipped from 2 variables with 3 constraints in the primal to 3 variables with 2 constraints in the dual.

The dual linear program is minimized at $a = 0$, $b = 3$, $c = 2$, with a value of 31. Therefore, the optimal values of the primal and dual objective functions are equal, and strong duality is confirmed to hold for this problem (as it does for all linear programs).

2. (20 pts.) **Spaceship.** Let x_1 be the number of oxidizer units provided for compartment 1, and define similarly x_2 for compartment 2 and x_3 for compartment 3. The probability that all units fail in compartment

1 is $(0.3)^{x_1}$, for compartment 2 is $(0.4)^{x_2}$ and for compartment 3 is $(0.2)^{x_3}$. The probability that all units fail in all compartments is $(0.3)^{x_1}(0.4)^{x_2}(0.2)^{x_3}$. The exponential function is non-linear, so we take logarithm (which preserves ordering of numbers) to get the linear program

$$\text{Minimize } \log(0.3)x_1 + \log(0.4)x_2 + \log(0.2)x_3$$

$$\text{subject to } \begin{cases} 40x_1 + 50x_2 + 30x_3 \leq 500 & \text{space constraint (cu in.)} \\ 15x_1 + 20x_2 + 10x_3 \leq 200 & \text{weight constraint (lb)} \\ 30x_1 + 35x_2 + 25x_3 \leq 400 & \text{cost constraint (\$1000)} \\ \log(0.3)x_1, \log(0.4)x_2, \log(0.2)x_3 \leq \log(0.05) & \text{reliability constraints (log scale)} \end{cases}.$$

(Note that the reliability constraints imply the non-negativity constraints that $x_1, x_2, x_3 \geq 0$.)

3. (20 pts.) **Maximum Flow.**

(a) Let $f_{u,v}$ be the flow on edge $e = (u,v)$. We can write the linear program as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{(s,v)\in E} f_{s,v} \\ \text{subject to} \quad & \sum_{(u,v)\in E} f_{u,v} = \sum_{(v,w)\in E} f_{v,w} \quad \forall v \in V - \{s,t\} \quad \text{(flow conservation)} \\ & f_{u,v} \leq c_{u,v} \quad \forall (u,v) \in E \quad \text{(capacity constraint)} \\ & f_{u,v} \geq 0 \quad \forall (u,v) \in E \end{aligned}$$

(b) Let f_p be the flow through s-t path p. Let the set of all s-t paths be P.

$$\begin{aligned} \text{maximize} \quad & \sum_{p\in P} f_p \\ \text{subject to} \quad & \sum_{p\in P:(u,v)\in p} f_p \leq c_{u,v} \quad \forall (u,v) \in E \quad \text{(capacity constraint)} \\ & f_p \geq 0 \quad \forall p \in P \end{aligned}$$

Note that the flow conservation constraint does not need to be specified explicitly when we consider entire paths at a time instead of individual edges.

(c) We multiply each of the capacity constraints by a dual variable $y_{u,v}$ ($|E|$ variables total).

$$\begin{aligned} \text{minimize} \quad & \sum_{(u,v)\in E} c_{u,v} y_{u,v} \\ \text{subject to} \quad & \sum_{(u,v)\in p} y_{u,v} \geq 1 \quad \forall p \in P \\ & y_{u,v} \geq 0 \quad \forall (u,v) \in E \end{aligned}$$

These dual variables can be interpreted as a length between s and t. The first constraint ensures s and t are separated by the end, just like they are in the residual graph after running Ford-Fulkerson.

(d) The capacity across the cut is the sum of the capacity of the edges that cross the cut in the direction from A to B. We need to construct a feasible solution to the dual with an objective function equal to that sum. The easiest way is to set $y_{u,v} = 1$ if $u \in A$ and $v \in B$, otherwise $y_{u,v} = 0$. Because every s-t path must cross the cut, the first constraint is satisfied. This assignment also makes the objective function equal to the capacity of the cut described at the beginning of the solution, as only edges crossing the cut in the correct direction have their capacities counted.

4. (20 pts.) **Cheap Max 3-SAT.**

(a) Given an instance of this problem, along with a proposed assignment that claims to satisfy at least a clauses by using at most b variables set to true, a polynomial-time certifier can verify the following:

(1) There are indeed at most b variables set to true (with a linear pass over the assignment).

(2) There are indeed at least a satisfied clauses (with a linear pass over the clauses, computing the value of each under the given assignment).

Both checks can be computed in polynomial time, and the certificate (the proposed assignment) takes up polynomial space as it assigns a single value to each of the variables. Therefore this problem can be verified in polynomial time, and it is in **NP**.

(b) Polynomial reduction from 3-SAT. Given an instance of the 3-SAT problem (a Boolean 3-CNF formula with n variables and m clauses), we can construct an instance of the Cheap Max 3-SAT problem as follows:

- Keep the Boolean formula the same.
- Set $a = m$, ensuring all clauses are satisfied.
- Set $b = n$, because we don't care how many variables we have to set to true.

Solving this instance of the Cheap Max 3-SAT will tell us whether the original 3-SAT instance is satisfiable. If it is satisfiable, the same assignment that satisfies the Cheap Max 3-SAT problem will also satisfy the original 3-SAT problem. This reduction is polynomial time because only the variables a and b must be set, a constant-time operation. Therefore, 3-SAT reduces to Cheap Max 3-SAT in polynomial time, and this problem is **NP**-Hard.

5. (20 pts.) **Hitting Set.**

(a) Given an instance of this problem, along with a proposed subset H that claims to hit every S_i while containing at most k elements, a polynomial-time certifier can verify the following:

(1) The proposed H is indeed size at most size k (by counting the elements in linear time).

(2) The proposed H hits each of the S_i. This can be done with a linear scan over each S_i asking "Is this item in H?". Even implemented with $O(n)$ searches, this is upper bounded by $|H|\sum_i |S_i| = O(|U|\sum_i |S_i|)$, which is polynomial on the input size since each S_i must be part of the input.

Both checks can be made in polynomial time, and the proposed H cannot be larger than U (even if it was, the first check would simply reject it). Therefore, this problem can be verified in polynomial time, and it is in **NP**.

(b) Reduction from Vertex Cover. Given an instance of Vertex Cover (an undirected graph $G = (V,E)$ and an integer k), we create an instance of Hitting Set as follows:

- Let $U = V$, the set of all vertices.
- Let $S_i = \{u,v\}$ for all $(u,v) \in E$. In other words, create a subset for each edge in the graph. Hitting a subset is analogous to covering an edge.
- Keep k the same.

Solving this Hitting Set instance also solves the Vertex Cover instance. Notice how, if every S_i is hit by some $H \subseteq U$, every edge would be covered by choosing the same subset of V in the graph. There is a one-to-one correspondence between vertices and items and between edges and subsets. Creating the hitting set instance takes polynomial time, as the inputs are largely the same. Retrieving the solution to the Vertex Cover from the Hitting Set is similarly easy, simply choose the vertices corresponding to items chosen in H. Therefore, Vertex Cover reduces to Hitting Set in polynomial time, and Hittitng Set is **NP**-Hard.

Rubric:

Problem 1, 20 pts

(a) 4 pts for correct plot

(b) 4 pts for correct linear program

(c) 6 pts: 2 for initial tableau, 1 per correct tableau step, 1 for correct answer.

(d) 6 pts: 4 for correct dual, 2 for the optimal variable values.

Problem 2, 20 pts
4 pts for the objective function, 4 pts per constraint.

Problem 3, 20 pts

(a) 5 pts for correct LP.

(b) 5 pts for correct LP.

(c) 5 pts for correct dual.

(d) 5 pts for correct variable assignment.

Problem 4, 20 pts

(a) 8 pts for NP argument (may be shorter, make sure they say what must be verified and how it's polynomial time).

(b) 12 pts for reduction (could be a different reduction, make sure they explain how it's polynomial time).

Problem 5, 20 pts

(a) 8 pts for NP argument (may be shorter, make sure they say what must be verified and how it's polynomial time).

(b) 12 pts for reduction (could be a different reduction, make sure they explain how it's polynomial time).